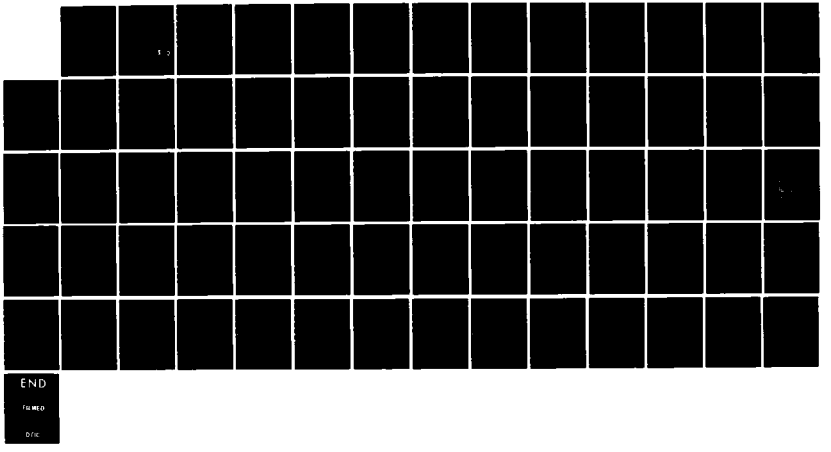
AD-A154 709 THE DOCUMENTATION ASSISTANT: AN INTELLIGENT SYSTEM FOR DOCUMENTATION..(U) ADVANCED INFORMATION AND DECISION SYSTEMS MOUNTAIN VIEW CA J S DEAN ET AL. 22 APR 85 1/1

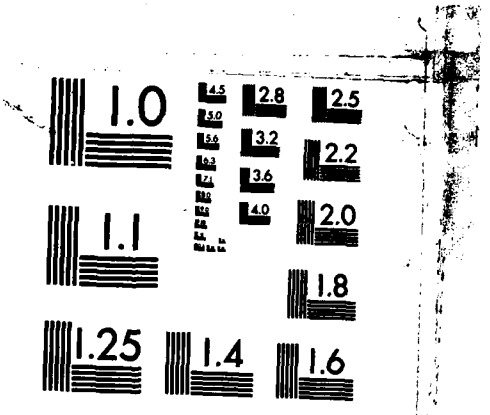
UNCLASSIFIED AI/DS-TR-1047-020-1 N00014-83-C-0444 F/G 9/2 NL



END

FILMED

DTIC



AI.DS
ADVANCED INFORMATION
& DECISION SYSTEMS

201 San Antonio Circle, Suite 286
Mountain View, CA 94040
(415) 941-3912

TR-1047-020-1

THE DOCUMENTATION ASSISTANT:

An Intelligent System for Documentation

Jeffrey S. Dean
Brian P. McCune
Susan G. Rosenbaum

April 22, 1985

Final Report for 1 July 1984 - 12 December 1984

Approved for public release; distribution unlimited

Prepared for:

Office of Naval Research
Department of the Navy
800 North Quincy Street
Arlington, Virginia 22217

The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Navy position, policy, or decision, unless so designated by other official documentation.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE AI/DS -	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR-1047-020-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Advanced Information & Decision Systems 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State and ZIP Code) 201 San Antonio Circle, Suite 286 Mountain View, CA 94040-1270	7b. ADDRESS (City, State and ZIP Code) Department of the Navy 800 N. Quincy Street Arlington, Virginia 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-83-C-0444
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT NO.
11. TITLE (Include Security Classification) The Documentation Assistant: An Intelligent System for Documentation	
12. PERSONAL AUTHOR(S) Dean, Jeffrey S.; McCune, Brian P.; Rosenbaum, Susan G.	
13a. TYPE OF REPORT Final 13b. TIME COVERED FROM 84/07/01 TO 84/12/01	14. DATE OF REPORT (Yr., Mo., Day) 85/04/22 15. PAGE COUNT 68
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES		
FIELD	GROUP	SUB. GR.
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Extended Program Model, Documentation Standards, Documentation, Software Engineering, Programming Environment, Intelligent Program Editor, Artificial Intelligence, Documentation Structure

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this document is to describe a semi-automated system for the documentation of computer software; this system is called the Documentation Assistant (DA). To motivate this system, a general discussion of documentation issues and problems is interwoven with descriptions of how the DA would address many of these topics. A feasibility assessment and plan for this approach is presented.

The Documentation Assistant project is a research effort at Advanced Information & Decision Systems. The purpose of this effort is to study advanced techniques which address one of the most pressing problems during the software life cycle: the process of documentation. Currently, the DA exists on paper only; one goal of the research effort is to develop a prototype version, which would be incorporated in the Intelligent Program Editor (another research prototype being developed at AI&DS under ONR sponsorship).

Additional keywords: systems engineering, artificial intelligence, AI

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☒ DTIC USERS ☐	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Robert B. Grafton	22b. TELEPHONE (Include Area Code) (202) 696-4713 22c. OFFICE SYMBOL Code 430

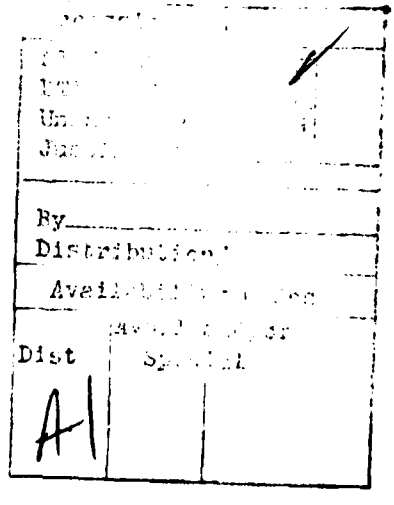
DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

CONTENTS

TABLE OF CONTENTS



CONTENTS

FIGURES

LIST OF FIGURES

1. OVERVIEW

1.1 PURPOSE OF THIS DOCUMENT

The purpose of this document is to describe a semi-automated system for the documentation of computer software; this system is called the Documentation Assistant (DA). To motivate this system, a general discussion of documentation issues and problems is interwoven with descriptions of how the DA would address many of these topics. A feasibility assessment and plan for this approach is presented.

The Documentation Assistant project is a research effort at Advanced Information & Decision Systems. The purpose of this effort is to study advanced techniques which address one of the most pressing problems during the software life cycle: the process of documentation. Currently, the DA exists on paper only; one goal of the research effort is to develop a prototype version, which would be incorporated in the Intelligent Program Editor (another research prototype being developed at AI&DS under ONR sponsorship).

1.2 PROBLEM

The software development and maintenance processes currently consume extraordinary quantities of resources. A great deal of this cost can be attributed to the loss of information and knowledge during the software life cycle [Dean-83]. As people work on software, they learn a great deal about it; much of this information is forgotten (as they move on to new things) or lost (as they change jobs).

The purpose of documentation is to provide a means for capturing information. Unfortunately, current documentation practices fall far short of being able to stem the loss of information, since documentation is treated as a separate (and often less important) activity. The result of this is that documentation is inadequate: it is often incomplete, out of date, and inaccurate.

Improving the documentation process can make a considerable impact on the software development and maintenance process. Clearly, the process of writing documentation is expensive -- it is not amenable to full automation, and doing it correctly is more work than doing it incorrectly. However, over the long term (and especially in the maintenance phase), the cost of good documentation will pay for itself many times over.

1.3 APPROACH

The Documentation Assistant addresses issues relevant at all stages of the documentation process, from requirements analysis in the beginning to maintenance in the end. However, as a starting point, the research described in this

document focuses primarily on that documentation which is written and maintained by programmers (i.e., in-line program comments and related documentation). This focus should not be construed to imply that these ideas are useful only to programmers. The capabilities provided by the Documentation Assistant will benefit all those who work with documentation.

The Documentation Assistant presents a unique approach that will alter the way people deal with documentation. It will provide:

- *Integration:* The DA will be part of the programming environment; it can be used just like other programming tools in the environment. There is no need to switch contexts in order to work with documentation.
- *Assistance:* The DA will help the programmer perform documentation tasks by providing both tools and structure. It will not automate documentation -- people are an essential part of the documentation process.
- *Intelligence:* To ensure that large quantities of documentation are kept up to date and consistent requires considerable knowledge about the documentation process; to do this without overburdening or interfering with the user requires an equal amount of knowledge about users and how they will use the system.

Not only does the DA provide new ways of working with documentation, it also provides different ways of representing documentation. Documentation will be:

- *On-line:* The computer is the home for all documentation. While a hard-copy form of documentation can be produced, the primary/original form is always on the computer.
- *Structured:* Rather than being viewed simply as text, documentation is recognized to have structure, and this structure is used to help guide the documentation process.
- *Traceable:* The dependencies between different documents (or different parts of a single document) will be represented by the DA.
- *Controlled:* The creation and modification of documentation will be governed by the DA, allowing the system to keep track of the documentation, and ensure that documentation is handled correctly.

The DA represents a paradigm shift in the way documentation is used, by treating documentation with the same care and formality which has been applied in the past only to code. By providing new tools for handling documentation and new techniques for representing documentation, the DA has the potential to significantly improve the production and maintenance of documentation.

1.4 FEATURES

The DA will provide the following features:

- *Integrated programming environment:* Documentation support is provided as part of the programming environment.
- *Structured editor:* Documentation is created and modified with an editor knowledgeable about the structure of documentation.
- *Documentation tied to programs:* Programs are explicitly linked to related documentation.
- *Navigation aids:* Interactive tools are provided for browsing, traversing, and searching documentation.
- *Document formatting:* Documents can be formatted using standard text formatting facilities.
- *Detection of outdated documentation:* Missing or outdated documentation is automatically detected.
- *Policy support model:* documentaTion policies, standards, and guidelines are explicitly represented in a parameterized model.
- *User preference model:* The user interface is based on parameterized information about user preferences.

1.5 SCENARIOS

To provide a better idea of how the DA might appear in use, this section presents a scenario of such a system in operation. The scenario is based on a programmer working in a maintenance environment, who is responsible for the maintenance of both code and documentation of a subsystem. The version of the DA described is imbedded in the program editor which the programmer normally uses.

Two caveats are in order. First, this scenario is hypothetical; there is currently no system that does any of this. Second, the interaction between a user and the system would, in practice, be primarily graphical; the DA will make use of graphics for both input and output. Unfortunately, there is no good way to show this here, and so the interaction between the user and the DA is presented in narrative form.

Referencing documentation:

The user is examining a program, trying to understand it in order to fix a problem. He brings the program up on the screen and sees a call to a procedure with which he is unfamiliar, so he uses the mouse to indicate that he is interested in that function. A pop-up menu of available documentation for that function appears. The user again uses the mouse to select the entry on the menu that corresponds to documentation on how the function works. A new window containing the requested information appears on the screen.

Tracing through documentation:

The documentation makes reference to another procedure which he doesn't know, and so the user follows a similar procedure, selecting the function (though this time the function reference appears inside documentation, and not inside code) and then selecting the documentation entry from the pop-up menu, giving him a display of documentation describing how this new procedure works.

Restoring context:

When he is finished reading this documentation, the user asks the system to pop back to where he was; the documentation for the second procedure disappears from the screen, and the documentation for the first procedure reappears. The user asks to pop back again, and the documentation again disappears, leaving him in the code where he originally started.

Creating documentation and code:

The user feels that he has an adequate understanding of the problem now, and so proceeds to start fixing things. He first adds some new lines of code to the part of the program that appears incorrect, but quickly realizes that he needs to define a new procedure to perform a calculation. He moves to an appropriate place in the code and starts the definition of the new procedure. Since he is writing a new procedure, the system displays a standard procedure header form on the screen for him to fill out as he writes the function. While he is working, he alternates between writing code and writing documentation.

Explanation by example:

There is one part of the documentation form that he does not understand, and so he selects that part with the mouse; when a pop-up menu appears, he selects the entry for sample documentation. The part of the documentation he was unsure of is now filled in with a sample of what this type of documentation should look like. This sample makes it clear to him how this field should look, and he then finishes that part of the documentation.

Reminder to finish documentation:

After finishing the new procedure, the user tries to pop back to the original program he was fixing. However, he has left out some important parts of the procedure header documentation, and so the system asks him if he wishes to write the documentation now. He does some of the required documentation, but then decides to leave the rest for later, and he tells the system he does not want to update the rest of the documentation right now.

Reminders to add new and update old documentation:

Now that he has written the new procedure, he is ready to finish the fix to the first function. He makes the appropriate code changes, and then tries to save the changes. The system prompts him for new documentation describing the changes he has made (note how he was prompted for the changes after he was finished, rather than after he made the first few changes). The system then prompts him with old documentation that might need to be updated. He realizes that some of this old documentation does indeed need changing, and so he updates this documentation.

Reminders to finish documentation:

After completing this documentation, the system then asks him to finish the documentation for the new procedure that he left incomplete. Since he would like to test the program before completing the documentation on that code, he declines to finish the documentation now, knowing that he will be reminded in the future that the documentation needs updating. He saves his work and exits the system.

1.6 GUIDE TO READING

Sections 2 through 4 represent a design plan for the DA; Section 2 describes the process and structure (and hence representation) of documentation; Section 3 addresses the issues of controlling documentation; and Section 4 presents user interface techniques. The feasibility of the DA approach is covered in Section 5. A plan for implementing the DA is presented in Section 6. Possible directions for future research are briefly discussed in Section 7. Section 8 is the conclusion, and references for this report are in Section 9. Appendix A is a reprint of a paper on a related effort, the Intelligent Program Editor. Appendix B is a reprint from another related effort, the RUBRIC information retrieval system.

2. THE DOCUMENTATION PROCESS

The Documentation Assistant is designed to provide intelligent assistance in all phases of documentation production and maintenance. We use the term "software documentation" to refer to all written pieces of information pertinent to a software system throughout its life cycle, including (but not limited to) requirements, specifications, design, design rationale, source code, in-line comments, test plans, test data, test results, modification history, problem reports, user manuals, operations manuals, and maintenance manuals.

A necessary part of any system purporting to provide intelligent behavior is a model of the process/environment in which the system functions. We break the process of documentation into three components:

- *structure of documentation*, i.e., the form of the documentation itself
- *context/state model*, which tracks significant events in environment
- *policy model*, which represents constraints on documentation such as standards, guidelines, and preferences

The following sections discuss these components in more detail.

2.1 THE NATURE OF DOCUMENTATION

The prerequisite for understanding the documentation process is to understand documentation itself. However, there are many views and opinions on what documentation is. The view taken by the DA treats documentation in terms of the following three components:

- *name:* Each piece of documentation has a name that can be used to reference it.
- *structure:* Pieces of documentation can be interconnected to form a structure.
- *attributes:* Each piece of documentation may have certain properties or additional information associated with it

The following sections cover these components in more detail, discussing representation techniques that will be used in the DA. The views taken here are partially motivated by research in the areas of semantic networks and object-oriented programming.

2.1.1 Name

Every piece of documentation has a name associated with it. The name of a document has the same usefulness and functionality as the name of a person: it can be used to refer to that document or person. When objects are to be dealt with as individuals, names are an obvious characteristic (objects which are dealt with as aggregates, on the other hand, do not need to be individually named; they can be named descriptively or procedurally). Names do not have to be meaningful, though in certain domain such as programming, it is desirable for the names to have some meaningful interpretation.

As with people, names may not necessarily be unique. When this happens, additional information is needed to provide disambiguation. For example, "John Smith who lives on Short Street" can be used to specify a person; "the Requirements Specification Document for Accounting Package" can be used to specify a document. Note that both of these specifications might be ambiguous in a global setting (e.g., "Short Street in which city?", "Accounting Package for which company?"), but it is necessary to specify only enough information for disambiguation with respect to some local context.

Names can refer to classes of documentation as well as instances. For example, "the user manual for the Emacs editor on TOPS20" is an instance of documentation; "the user manuals for editors" is a class of documentation describing a set of documents. From a representational viewpoint, classes and instances are treated identically.

2.1.2 Structure

Structure is the way in which pieces of documentation are woven together. It is important for those who read documentation as well as for those who write it; it is also a logical model for the representation of documentation used internally by documentation systems. Knowledge of document structure can assist in many parts of the documentation process, including:

- *creation:* Structure can guide the creation process by making sure that documentation is assembled properly.
- *searching:* Instead of scanning the entire document, structural knowledge allows searching to be limited to the relevant (sub)section.
- *understanding documentation:* The connection of high level abstractions to lower level details provides traceability that makes it easier to understand documentation.

It is natural for different kinds of documentation to be structured in a variety of ways; possible structures include sequential/linear (one dimensional), hierarchies/trees (two dimensional), and graphs/networks (n-dimensional). For

scenario ("if it looks like it has changed, then it has"), this estimate can be carried along to the next step of the process.

3.2.2 The State of the Programmer

The preferences of the user are related to the context model, thus allowing the DA to make decisions based on user preference with respect to a given state. For example, a user might specify that he wants to update procedure level documentation whenever he edits and then leaves the procedure; another user might want to do the updating only at the end of the edit session; and another user might not want to do updates until the code has been tested and is known to be working.

Thus, via the preference mechanism, users can control when they will be prompted for documentation creation/update. Since the DA will be capable of interrupting the user to ask for documentation, this type of control is important to prevent the system from getting in the way of the user. The quantity of interruptions is also moderated by the mechanisms that allow unimportant changes to be ignored or put aside.

3.2.3 A Rule Base for Controlling Documentation

The process of asking the user to update documentation is rule-based. The rules are based on various criteria mentioned earlier, such as importance of documentation, likelihood of semantic change, preference of user, state of user, etc. The set of rules is not fixed; it can be tailored to specific environments.

The rest of this section is an example of how a rule-base might be used. Rather than presenting specific rules, however, we present rule specifications, which describe a class of rules. These specifications are in the form of functional mappings; they map the space of the left side (the Cartesian product of the independent variables) to the right hand side (the dependent variable).

Suppose it is necessary to determine when to notify a user about a particular documentation object (attached to some segment of code) being out of date. To answer this question, start with rule (1), whose right hand side can answer this question. To evaluate this rule, it is necessary to evaluate all the dependent variables on the left hand side. That is, to determine when to notify the user about out of date documentation, there are three things to look for: the probability that the documentation is really out of date, the importance of the documentation for that particular part of the program, and the user's preferences.

$$(1)\ \left\{\begin{array}{c}probability\ of\\ documentation\\ out\ of\ date\end{array}\right\} \times \left\{\begin{array}{c}importance\\ of\ link\end{array}\right\} \times \left\{\begin{array}{c}user\\ preference\end{array}\right\} \rightarrow \left\{\begin{array}{c}when\ to\\ notify\ user\end{array}\right\}$$

administratively and technically, it is difficult for people to become conversant with (and able to apply) all appropriate policies. By providing tools that help apply policies, it becomes more likely that policies will be followed. Second, policies are sometimes neglected because they are difficult or cumbersome to practice. For example, policies requiring documentation to be kept up to date with the latest code require programmers to continually keep track of change and of the documentation affected by those changes. Documentation tools can help alleviate this problem by providing support for the more mundane documentation chores and thus reduce the effort required to do things correctly.

3.2 TECHNICAL ASPECTS OF DOCUMENTATION CONTROL

The process of keeping documentation updated is controlled by several factors: the state of the documentation and code, the state of the programmer, and a rule-base for controlling documentation.

3.2.1 The State of the Documentation and Code

The DA will keep track of the state of the documentation. By virtue of the Extended Program Model on which the IPE and DA will be built, it is possible to maintain a better idea of what is happening to code and documentation than has previously been possible. As a program is changed, the IPE will have the capability to determine if the *meaning* of the program has changed.[1] This is possible because of the multiple program representations provided by the EPM. For example, if a program were textually reformatted but unchanged, it is easy to determine that the program semantics are unaffected, since there is no change in the syntactic structure; if a statement were inserted into the middle of a cliche, it might be determined that the cliche didn't change by examining the flow representation and realizing that the graph of the cliche was disjoint from the graph of the new statement.

At the coarsest level, the EPM will be able to determine if an object has really changed. Detected program changes can be propagated back to documentation (following the links that connect the documentation to the EPM described earlier). At the next level, it is possible to draw associations between certain semantic changes and documentation objects. For example, there is an obvious connection to the syntactic object *parameter_list* and the documentation object *parameters*. If a procedure changes, but the parameter list does not, then there is no need to change the parameters documentation, even though other documentation linked to that procedure might need to be changed.

The idea of program change is not a boolean decision. There may be times when a likelihood of semantic change (and propagation to documentation) can be assessed on a probabilistic scale. Instead of always assuming the worst-case

---

1 Actually, it will make a probabilistic hypothesis about changes in semantics, since it is impossible to detect arbitrary semantic changes.

suggestion). There will always be events that fall on the vital end of the scale; there is simply no way of avoiding certain things, and when it comes to the issue of vital policies, preference must give way. On the other hand, to gain acceptance, it is crucial to avoid annoying the user and continually overruling his preferences. Luckily, there are many events that do not fall at the extreme end of the scale, and in these cases, there are several techniques for working out compromises.

- *schedule negotiation*: If policy requires certain documentation, but doesn't say when it is required, it would be possible to make a compromise with a programmer whose preference was not to do the documentation. The compromise is basically this: the documentation does not have to be done now, but it has to be done at some definite time in the future (e.g., before the software is released). In this case, longer term policy needs are overridden by shorter term preferences. By warning the programmer that there is a policy requirement that must be met, there may be time to allow this information to sink into the programmer's mind; knowing about this future requirement, the programmer might even rearrange things (e.g., do some planning) so that the documentation will be easier to do.

- *balancing policy against preference*: If there is some flexibility in policy, it is possible to weigh policy against preference. A user can associate a degree of importance to preferences; this can be modified as the user desires. As choices/compromises are made by the system, the user can adjust preferences to achieve desired results.

- *following policy to the letter*: There may be some flexibility in policy that will allow adjudicating in favor of preference over policy. For example, if policy required a document, but certain parts of that document were not as important, it would be possible to avoid those sections if the user were so inclined.

At first glance, these compromises may be viewed as a method for allowing a programmer to avoid responsibility. However, these techniques serve an important role in making the DA a system that people will want to use. We believe that providing a means for balancing policy and preference will help achieve acceptance and will not be abused.

Thus, we believe that programmers are generally willing to follow policy;[1] however, they often need help in applying policy consistently and intelligently. This philosophy is based on two observations from our studies of software maintenance environments. First, policies are often neglected because of ignorance. Given the complexity of programming environments, both

---

1 This is a *necessary* assumption for any policy-based tools; if programmers must be *forced* to adhere to policy, then there is a much larger problem that tools alone cannot solve.

3. CONTROLLING DOCUMENTATION

The discussion so far has focused primarily on documentation itself and how the DA handles it. We now turn to how the DA will "handle" the user. This section focuses primarily on the issues of maintaining documentation in a "proper" state; the next section looks at the user interface issues.

3.1 POLITICAL ASPECTS OF DOCUMENTATION CONTROL

To keep documentation in a proper state, it is first necessary to define just what the proper state is. The idea of a proper state is relative to each documentation taxonomy; what is good practice in one environment may be forbidden in another. Of the three components that constitute the documentation process, it is the policy portion that really determines what is good and what is bad. Policy says what things should be, what things might be, and what things should not be. By definition, if policy is followed, then things will be in a proper state.

Keeping documentation in a proper state is only half the picture; the other half is helping the user. A documentation system should not be the master that controls the people who use it. Just the opposite: the users should control the documentation system. The only way to do documentation correctly is to have the support and help of the users. A documentation system cannot possibly work correctly if people refuse to use it or thwart its activities. Thus, there is a delicate balance that must be achieved: on one hand, there are policies that, within certain limitations, specify how things are to be done; on the other hand, people have their own ideas about how to do things.

The balance can be achieved by providing a policy-driven mechanism that takes great care in trying to accommodate individual preferences. There is often more compatibility between policy and preference than might be evident at first glance. Compatibility may be in the form of non-interference, where policy and preference do not conflict; it may be in the form of compromise, where there are mutually satisfiable alternatives; or it may be in the form of goal revision, where the original plan is modified. The DA will try to traverse this tightrope, simply because it is the most reasonable path; it is unrealistic to expect to change either policy or preference to suit the needs of a documentation system.

Achieving this goal is not an easy problem. The DA will *not* contain built-in mechanisms that will automatically solve the tension between policy and preference. Rather, it will provide a framework; in any particular environment, the ability of the system to achieve a proper balance rests in the hands of the individuals who create the documentation taxonomy.

The key idea behind achieving this goal is to balance the policy requirements, which are generally longer term, against the user preferences, which tend to be shorter term. When dealing with policy, the DA can choose where on a spectrum of importance any particular issue falls. The spectrum ranges from vital (the policy must be followed) to inconsequential (as in the case of a

Since it is rare for programmers to work under the constraints of a single set of policies, it is expected that there will be a great deal of interconnectivity between different levels of policy in the policy model. In most organizations, there is an apparent hierarchical structuring of policies. For example, there may be a set of documentation standards for the entire Department of Defense. The Navy may have documentation standards which augment the DoD standards. A command in the Navy may in turn have its own standards; a programming organization under that command may have its standards. Finally, a programmer may have his own preferences for things not specified by any of the standards. In this hierarchy of standards, lower level standards generally refine the higher level standards, not replace them. However, this need not always be true; organizations may have permission to override standards at a higher level.

Looking at content alone, there is a great deal of similarity between procedures, standards, guidelines, and preferences. They all specify a way of doing things; it is primarily in importance or necessity that they differ. Thus, the DA can represent policies in a uniform way, in terms of constraints that apply to the documentation process. However, the importance of the constraints is represented separately. For example, suppose there is a constraint specifying that module level documentation include a revision history. If this is specified by a standard, then the constraint is vital; if this is specified as a guideline, then the constraint is recommended; if it is specified as a preference, then the constraint is weak.

what the programmer is doing.

2.3.2 Tracking the Documentation and Code

The next step in tracking the programming process is to keep tabs on the status of the documentation and the code. There are a number of states a software system may be in, including: preliminary design, design, development, debugging, unit testing, system testing, beta testing, and released. Knowing the state of a system as a whole does not mean that all the components of the system are in the same state; each component, subsystem, etc., must also be tracked. Tracking the documentation may be a somewhat easier job than tracking the programmer; since states do not change all that often, it is not unreasonable for the programmer to tell the system the state of documentation and code.

However, the system is capable of tracking documentation or code that has changed. This information is used to provide a more accurate assessment of the state of documentation and code; it is stored in the documentation database (and is not lost between sessions). For example, if the system is in the "released" state, and the code is modified, it should be inferred that the state of the system has changed.

2.4 DOCUMENTATION POLICIES

Thus far, we have talked about two dimensions of the documentation process: the nature and structure of documentation, and the tracking of programmers and documentation. The final dimension necessary for the DA concerns the choice of what and how documentation is created and updated.

The decisions as to what documentation should be written, how it should be written, how it should be updated, etc., are not determined just by the programmer. There are usually administrative procedures that specify what documentation is required, standards/guidelines specifying/suggesting how to write documentation; and only if there are decisions that are unspecified is the programmer allowed to follow his own preferences. These procedures, standards, and guidelines are collectively referred to here as documentation *policies*.

The DA will maintain models of these policies, which it will use to guide the process of creating and updating documentation. Policies will be represented in a structured fashion; connections between policies will be explicitly noted. Thus, it can easily be determined if part of one policy refers to or overrides (or conflicts) part of another.

The explicit representation of policy is meant to provide for easy accommodation of different policies. While different organizations may have similar (or overlapping) policies, it is rare for separate groups to have identical sets of policies. The policy model factors out this information, ensuring that knowledge about policy is not hardwired into the DA. To make the DA support the policies of a different organization, it would only be necessary to modify the policy model, rather than rewrite the DA itself.

In order to do this intelligently, it is first necessary for the DA to understand what the programmer is doing. The context which we focus on here is the editing context, where the programmer may be creating, modifying, or reading programs. Since the DA will be tied to the Intelligent Program Editor, this choice of context is logical. From a larger perspective, the context should certainly include programming activities outside the scope of the editor; it might even include activities outside the scope of the computer system (e.g., "what time does the programmer leave for the day?" or "when does the programmer go on vacation?").

There are two basic approaches for determining what the programmer is doing (and where he is doing it):

- *announcement*: The programmer "announces" to the system what he is doing (i.e., the user does the work).
- *inference*: The system watches the individual actions the programmer takes, and tries to piece them together into a plan to provide a larger model of what the programmer is doing (i.e., the system does the work).

The first approach is cumbersome, but nonetheless useful because there are certain times when the only way to figure out what is happening is by asking the user. The second approach requires a good deal of intelligence on the part of the system. To achieve this level of understanding, it is first necessary to build a library of plans that describe common sequences of actions. Then, techniques for sorting through a large number of plans in search of the plausible one(s) are necessary. Finally, it is necessary to determine which plans really fit what the user is doing. Based on current technology, this approach may be rather expensive.

However, there is another alternative that is essentially a combination of the two. Suppose that an editor had certain functions that, when used by the programmer, would give insights as to what was currently happening. That is, these functions would be designed so that when they are invoked, the editor would be able to guess fairly easily and reliably what the programmer was doing. For example, in the Emacs editor, there is a command for compiling code without leaving the editor; use of this command is an indication that the programmer thinks that the code is complete (at least, complete enough to run). As another example, imagine an editor that has a special command for editing procedures; when the command is invoked with a procedure as an argument, only that procedure is displayed on the screen, and editing continues on that procedure until the programmer gives a command to edit another procedure. This command would allow the editor to infer what function the programmer was editing.

Thus, by providing commands that work on semantic units, instead of textual units, an editor may be able to infer a great deal about what the programmer is doing. By virtue of the Extended Program Model provided by the IPE, the IPE is in a good position to provide extended commands that are based on the syntactic or semantic structure of programs, thus providing better clues as to

a single monolithic database), an incremental approach to EPM development can be taken. Databases and tools for their manipulation can be developed and then integrated into the EPM. Separate databases for the EPM also mean that each representation can be stored in the most appropriate type of database. For example, the database representation for text (i.e., a linear program representation) will be quite different than the representation for a syntax tree (i.e., a two dimensional representation).

Thus, the architecture of the EPM provides a natural way of adding additional representations. In the case of documentation, a new database supporting documentation structure and operations would be added. The documentation database will provide three basic components: documentation objects (representing the documentation itself), attributes (representing properties of documentation), and relationships (representing the connections that tie documentation objects together).

2.3 THE CONTEXT MODEL

The next step in building an intelligent documentation tool is to build tools for understanding the *process* of documentation. In order to manipulate documentation properly, it is necessary to understand what the user is currently doing and the current state of the documentation. The focus here is narrowed to the process of documentation from a programmer's point of view; hence, the emphasis will be primarily on in-line documentation. However, these ideas are not restricted to in-line documentation; given that documentation is on-line, these ideas can be applied to all phases of the documentation process.

The *Context Model* is a part of the DA that deals with keeping track of all that is happening in the programmer's environment. This includes keeping track of what the programmer is doing (e.g., writing code, debugging code, fixing documentation) and keeping track of the status of code and documentation. The Context Model has two components, a passive component, in which all relevant information is stored, and an active component, that is responsible for collecting and maintaining the information in the database. The Context Model is an integral part of the DA, and is not directly visible to users; thus, most users would not even be aware of its existence as a separate component.

2.3.1 Understanding What the Programmer is Doing

One of the key ideas behind the DA is to help the programmer write and update documentation without being intrusive. If the system gets in the way of the programmer, he will eventually turn it off or ignore it. On the other hand, in order to ensure that documentation is kept up to date, it may be *necessary* for the DA to intrude on the programmer. The DA will be an active partner in the documentation process, and, unlike conventional tools, it will be capable of taking the initiative and asking the programmer to do something.

documentation that hide much of this detail.

The DA will be able to provide this documentation structuring because it will build on the Intelligent Program Editor (IPE), which provides a rich environment for program manipulation. In particular, the Extended Program Model part of the IPE will provide the mechanisms necessary for this cross linking.[1]

2.2.1 The Extended Program Model (EPM)

The Extended Program Model can be thought of as a database that maintains multiple representations of programs. Currently, there are plans for six representations: text, syntax, flow, segmented parse, cliche, and intentional aggregate [Shapiro-84]. The EPM will maintain consistency among these representations, and the IPE will allow any of these representations to be directly viewed and manipulated. For example, if an IPE user were examining the syntactic representation and made a change, the corresponding change would be made to the text and other representations.

The representations in the EPM are linked together, allowing the IPE to map between objects in different representations. By adding documentation to the EPM as an additional representational level, the DA will be able to use the EPM to connect documentation to program objects as well as to other documentation.

Thus, by building on top of the EPM, the DA will be able to integrate documentation with code. Moreover, the DA can make use of the various representations in the EPM, enabling documentation to be linked to code at any level, and not just the textual level. For example, the documentation for a procedure header might be linked to the syntactic unit corresponding to the entire procedure, instead of (as in current practice) placing the documentation text just above the procedure in its text form. This means that the binding between the code and the documentation is increased; if, for example, the procedure were to be removed, it would be clear that the documentation should also be eliminated.

The DA should also be able to make use of the version control facility planned for the EPM. Since the documentation is so closely tied to the EPM, it is anticipated that the EPM's version control mechanism will also be able to provide version control for documentation. Because of the linkages between documentation and the EPM, the version control of documentation will be intimately tied to the version control of the code itself. It will not be possible to lose synchronization between the code and the documentation, as the separation between code and documentation itself will be blurred.

While the EPM will provide a uniform view of different program representations, it is internally composed of a number of databases, linked together to provide required connectivity. By building upon separate databases (instead of using

---

1 The Intelligent Program Editor (and its Extended Program Model) is being developed under a research contract sponsored by ONR.

objects, and providing mechanisms for connecting and describing those objects, it becomes feasible to build tools for documentation manipulation.

The decomposition of documentation should be considered an internal mechanism for the DA; the user does not see his documentation shredded into thousands of objects and thrown together into some incomprehensible structure. Documentation structure is retained using the relationships mechanism described earlier. One can think of a process where documentation is divided up into objects, each object is labelled, and then connections between the objects are drawn.

The next step towards elevating the treatment of documentation is the storing of all documentation objects in a database, separately from all else. The database provides two functions: it provides a convenient method for storing large quantities of documentation objects, and it provides a means for regulating the modification of documentation. The means of this regulation is simple: since documentation is no longer treated as just a text file, there is no way of directly going into the database to edit the text of a document. The reason for this is to preserve and control the structure of the documentation. If documentation is decomposed into component objects, it does not make sense to allow unrestricted editing of documentation, since this might destroy the structure. Instead, the DA will allow editing of documentation objects in a controlled context, so that structure is preserved. Moreover, this level of control allows the status of documentation to be tracked; when documentation is updated, it is easy to determine which part of the documentation was modified.

In addition to the structure inherent in the documentation itself, the DA provides a way of directly linking documentation to the program code. This is very different from the traditional practice of in-line comments. In-line comments have no formal standing with respect to most programming tools, which generally discard comments during parsing; even when comments are preserved, there is no formal way of associating comments with a given piece of code. This is a task that is very easy for a human user but intractable for a computer. If a user sees a comment next to or on top of a piece of code, the user can generally make the assumption that this comment refers to the adjacent code. Unfortunately, this is an assumption that a computer, unlike a human user, has no way of verifying. Imagine that a comment refers to code which is then changed (or even worse, deleted). What happens to the comment? By providing a formal linkage between code and documentation, the impact on documentation of changing or deleting code can be assessed.

With the same mechanism, documentation that is not normally considered in-line, such as specification and design documents, can be linked directly to code. This provides the ability to look at a piece of code and then trace back to the design or specification; similarly, one can look at the specification and trace through to the code which implements part of that specification.

The result of decomposing documentation into objects and linking them directly to the EPM results in a plethora of objects and links, possibly to the point of unmanageability for the human user. However, this structure is meant only for the DA; the user interface provides higher level access mechanisms to

desired result. For example, to determine the importance of a particular item of documentation, the DA might look to see if there is an *importance* attribute associated with that item; if not, then the DA might try to see if there is some general rule about the importance for all documentation of that type; and if that fails, the DA might try to find similar documentation objects and see what their importance is. It is possible that the inferencing mechanism will fail entirely to come up with the answer;[1] in this case, the DA could either make a worst case assumption, or could try another approach.

- *automated analysis:* Some attributes might be determined by an analytic routine specifically designed for this purpose. For example, if readability were a document attribute that was needed for some purpose, and a document did not have that attribute, then a readability metric might be applied to the document to determine the appropriate value for the attribute.

- *human analysis:* The human user can be called upon to do the job if necessary. This should be a last resort, since the user should not have to bother with details that the computer could figure out.

2.2 THE REPRESENTATION OF DOCUMENTATION

One of the primary differences between the DA and current documentation tools is that the DA treats documentation as a *first class citizen.* This means that documentation is not considered unstructured text, simply appearing in manuals or in-line, adjacent to program code. Documentation has both properties and structure; one should be able to point to a piece of documentation and say "What kind of documentation is that?" or "To what does this documentation refer?".

The DA will use several techniques to elevate documentation to this level. The above discussion on documentation structure loosely referred to "pieces" of documentation. Pieces of documentation are formally called documentation *objects.* A documentation object has a name, attributes, and relationships with other documentation objects as well as relationships with parts of the program code.

The purpose of decomposing documentation into a set of objects is to provide a handle for the computer-assisted manipulation. While it may be easy for people to look at documentation and intuit structure and meaning from it, this is an intractable problem for the computer. By breaking the documentation into

---

1 This might happen if the answer was not determinable from the knowledge base; it might also happen if the inferencing engine was not using the necessary strategies to find the answer.

At this point, there is an interesting observation that can be made about the documentation taxonomy. In a certain sense, names, relationships, and attributes are themselves a form of documentation. One can think of documentation as describing some system, while names, relationships, and attributes are the next level up, describing the documentation itself.

2.1.5 Classifying Documentation

The Documentation Taxonomy provides a mechanism for talking about documentation, but in order to use this mechanism, it is necessary to classify documentation with respect to the taxonomy. To classify documentation, it is necessary to decompose the documentation into its component pieces, and then determine the name of each piece and its relationship to other pieces. There are basically three mechanisms the DA will employ for doing this:

- *definition-time analysis:* When documentation is created using the DA, it will be immediately classified; the decomposition, name, and structure is implicit in the interface for most documentation (i.e., the documentation is entered in a form that is easily decomposed). This makes classification primarily a definition-time operation (as compared to a run-time operation), since the bulk of the work is done when the taxonomy is initially defined.

- *run-time analysis:* If documentation is not created via the DA, it is much more difficult to classify. It may be possible to provide automated tools to segment and classify documentation items. Unfortunately, unless a fairly strict set of documentation guidelines have been followed, this method is both weak and prone to errors.

- *human analysis:* The alternative to automatic post-analysis is human analysis. This is the most arduous technique of all, but it is more reliable than run-time analysis.

Determining attributes calls for slightly different mechanisms because attributes are used differently. Since the DA needs the decomposition, names, and relationships of documentation to construct documentation networks, this information is necessary just to get the documentation into the DA system. Attributes, however, are not essential until the associated documentation is actually manipulated. When it is necessary to determine attributes, there are several techniques (analogous, but not identical, to the above techniques):

- *automated inference:* Attributes can often be logically inferred using other information in the DA. Inferencing means that by ascertaining a set of logical premises, one can make certain deductions. In the simplest case, inferencing is simply looking up the fact in the knowledge base. More often, inferencing involves a chain of implications which lead to the

programming languages, the semantics of attributes such as type will not be built into the interpretation mechanism; instead, there will be support for the declarative specification of these properties.

The choice of attributes, and the values which attributes may take on, is partially dependent on the environment. While there are some attributes, such as type, which are always needed, there are other attributes that may not be needed. Moreover, even required attributes vary in the values which they can have; the possible values for the type attribute will vary across environments.

2.1.4 The Documentation Taxonomy

The notions of name, relationship, and attribute provide a means for talking about documentation in general. A *Documentation Taxonomy* is a set of names, relationships, and attributes that can be used to talk about the particulars of documentation.

The Documentation Taxonomy provides a basis for reasoning about documentation. The structure provided by the relationships and attributes allows the recording of many kinds of information about documentation. However, this information about documentation need not be complete. By applying inheritance rules to documentation structure, it may be possible to infer facts about documentation objects without having that information explicitly represented. For example, to determine how to format a documentation object, the first thing to check is to see if there is an attribute for that object which talks about formatting specifications. If not, then the parent of the object might be checked to see if it has any formatting specifications. If the parent doesn't have the attribute, then the search continues up the hierarchy, until an ancestor with the appropriate attribute is located.[1]

One of the more challenging aspects of building a system to manipulate documentation is that there are so many ways of using documentation. Note that in the previous discussion of documentation, there were no absolute statements about documentation requirements; for any particular application and environment, the particular names, relationships, and attributes will vary. While there may be commonality between different environments, the design of the DA recognizes this as an option rather than a given.

The goal of the documentation taxonomy is to provide a descriptive (rather than prescriptive) framework, allowing the DA to be tailored to particular environments. In many cases, taxonomies may be the same for different environments; certainly, parts of taxonomies will always be common to all environments. The process of building up a taxonomy for a new environment might be comparable to the process of moving an expert system to a different task in the same domain.

---

1 There are different techniques for determining the value of missing attributes; thus, there must be rules to determine which attributes can be inherited and which must be determined in some other way (such as asking the user).

example, hierarchical structures are common for textual kinds of documentation, such as manuals. Unfortunately, most existing systems that provide document structuring capabilities (such as word processors, mail programs, and documentation browsers) force documentation to fit into one particular type of structure. Because it is based on a more general mechanism for creating different types of structures, the DA will have considerably more flexibility.

In the DA, the idea of a *relationship* is used to capture structural information. A relationship is a connection among *n* (usually two) pieces of documentation. Connections themselves have names, which indicate the type of relationship. By providing connections with names, a diverse set of structures can be supported. Examples of connection names are: *contains* ("a procedure header *contains* the name of the procedure"), *references* ("the Program Description Document *references* the Program Design Specification"), *depends on* ("the Computer Program Test Report *depends on* the Computer Program Test Specification"), etc.

For example, if a particular procedure header contains information on the author of that procedure, then the relationship *contains* will hold between the procedure header documentation and the author documentation. Relationships can be among documentation *classes* as well as documentation instances, thus providing a way of specifying that all objects of some class have a certain relationship with all objects of another class (i.e., inheritance). As another example, if all procedure headers contain information pertaining to the date the procedure was written, then the relationship *contains* will hold between the class *procedure_header* and the class *date_written*.

2.1.3 Attributes

Attributes of documentation are descriptions or properties that pertain to the documentation. An attribute refers to a particular piece of documentation. There are many attributes that one might use to describe documentation, such as purpose, readability, completeness, correctness, quality, or currency. Attributes can be created and accessed by users as well as by documentation tools.

Attributes are easily represented as *property-value* pairs. For each piece of documentation, there can be an arbitrary number of property-value pairs. If the DA were posed a question of the form "What is the *status* of this piece of documentation?", it would attempt to answer this by checking the value of the *status* property. For example, if a program comment were added to clarify a bug fix, there may be an attribute *purpose* which has the value "clarifies the bug fix"; if a program comment were written by a particular programmer, there may be an attribute *author* which has as its value the name of the programmer.

There are certain attributes that are required for all documentation objects. For example, the *type* attribute is appropriate for all objects, regardless of environment. As in programming languages, the type attribute is a name which has certain semantic associations that characterize the referenced object. Type is an important vehicle for talking about documentation. It conveys information and expectations to anyone who knows the definition of the type. But unlike

Rule (2) says that the likelihood that documentation is out of date can be determined by the type of the documentation object, the type of the program object, and the type of the change that was made to the program object. The first two clauses can be determined by lookup; the other clause must be determined by the EPM.

$$(2)\ \left\{\begin{array}{c}type\ of\\ documentation\\ object\end{array}\right\} \times \left\{\begin{array}{c}type\ of\\ program\\ object\end{array}\right\} \times \left\{\begin{array}{c}type\ of\\ change\end{array}\right\} \rightarrow \left\{\begin{array}{c}probability\ of\\ documentation\\ out\ of\ date\end{array}\right\}$$

There are two more clauses from rule (1) that need to be evaluated. The importance of a link can be determined by evaluating rule (3), which says that the importance of a link can be determined by looking at the type of documentation object and the type of program object. For example, if the program object is a procedure and the documentation object is a procedure header, then the importance is "high".

$$(3)\ \left\{\begin{array}{c}type\ of\\ documentation\\ object\end{array}\right\} \times \left\{\begin{array}{c}type\ of\\ program\\ object\end{array}\right\} \rightarrow \left\{\begin{array}{c}importance\\ of\ link\end{array}\right\}$$

Finally, the user's preference can be determined by evaluating rule (4), which requires evaluation of the type of documentation and program object (already evaluated by the above two rules) and the state of the context model, which can be looked up.

$$(4)\ \left\{\begin{array}{c}type\ of\\ documentation\\ object\end{array}\right\} \times \left\{\begin{array}{c}type\ of\\ program\\ object\end{array}\right\} \times \left\{\begin{array}{c}state\ of\\ context\\ model\end{array}\right\} \rightarrow \left\{\begin{array}{c}user\\ preference\end{array}\right\}$$

This process of starting from a goal and recursively searching through a rule-base in an attempt to find a way to reach the goal is known as *goal-directed inferencing* or *backward chaining*. As values are determined for each clause, the values are passed back up, until the goal is reached. Since the proposed evaluation scheme for the system is based on a multi-valued logic, answers can be somewhere between "yes" and "no". Thus, reaching an answer may require the selection and application of cutoff levels.

4. THE USER INTERFACE

Understanding and manipulating documentation is one half of the DA; the user interface is the other half. The phrase "Documentation Assistant" is used to connote a partnership between the computer and the user. The human is an essential part of the documentation process; the purpose of the DA is to help, not replace, the user. To provide this help to the user, the DA must provide an interface that is intelligent and user friendly.

The most important aspect of the interface provided by the DA is the integration of the documentation into the programming environment; in particular, the DA will be built on the Intelligent Program Editor, and will allow natural access to documentation through the editor. There should be no need for the user to switch context in order to manipulate documentation.

The IPE will provide a window-oriented user interface. It will present a display consisting of a series of windows, allowing for the user to know the state of the system at all times through the visual representation on the screen. The documentation and code under scrutiny can be displayed in a multiple number of views. The user can move easily among the different views by moving between (or creating new) windows.

Multiple modes of interaction will be provided by the IPE. User input can occur through both *menu item selection* and *keyboard input*. Each of these modes will be available simultaneously; the user can use whichever is more convenient. The system will be highly customizable, allowing the interface to be modified to suit the preferences of individual users.

In addition to the IPE functions, made available by building the DA on top of the IPE, the user interface will provide a number of documentation-specific functions. The rest of this section will describe some of those functions.

4.1 VIEWS

Previous discussion on documentation representation described a way of splitting documentation into small chunks or objects. This provides a natural way for a computer system to manipulate documentation. However, it provides a most inconvenient model for people, who may prefer to deal with documentation in larger units. The *view* mechanism of the DA provides a user level mechanism for handling documentation.

A view is like a template: it provides a frame for displaying a set of documentation objects together (Figure 4.1).

Views are named entities. For example, to see general information about a module, a user might ask to see the *Module Info* view for that module (Figure 4.2). There may be many views for one particular object; a user might ask to see the *Algorithm Info* view for the same module (Figure 4.3).

view-name	
link-name	*link-value*
link-name	*line-value*
...	...

Figure 4-1: The Structure of a View

Module Info	
Name	sort
Author	Joe Ada
Date	August 1984
Revision	2.1

Figure 4-2: The Module Info View

Algorithm Info	
Name	sort
Keywords	sort, bubble sort
Description	bubble sort algorithm using ...
Parameters	integer array to be sorted
Reference	Knuth, vol. 3, page xxx

Figure 4-3: The Algorithm Info View

There is no one set of views that will be adequate for all environments, since the specification of views is based on the taxonomy itself. Different libraries of views will need to be provided for each environment; but, since views are primarily meant to be a user interface tool, the DA will also permit the creation of new views by individual users in order to suit their own needs.

In the above examples, views were used to examine existing documentation. Views can also be used for creating or modifying documentation. To document a module, a user might ask the DA to create a *Module Info* view for that module. The system would display the template (with known values, such as date or author, automatically filled in); creation of the documentation then becomes the process of filling in the template. Such a technique provides a way of structuring the documentation creation process. If the user wants to set up general module information, the view makes it clear what should be included in that information. It also provides a way of analyzing documentation: when filling in a template, the decomposition of the documentation into objects is obvious (since each slot in a view is a separate object).

To go a step further, if the user needs some assistance in writing documentation, it would be possible to display an example of how that documentation might look. This sample could be a fake version, created specially for this purpose, or it could be actual documentation for another part of the system. For documentation that is standardized, this information should be readily available, since standards generally give examples or descriptions of how documentation should look.

4.2 INTERACTION CONTROL

It has been customary for the programmer to have control over the programming environment; actions are initiated on request only. This mode of interaction is known as *user initiative.* User initiative places the responsibility for all actions on the user. In complex environments, there may be too many factors (such as programming standards, guidelines, administrative procedures, module interconnectivity, etc.) for the programmer to deal with in any reasonable fashion. The alternate approach is called *system initiative*, wherein actions are initiated by the system without user intervention. System initiative is particularly useful in complex environments because computers are good about keeping track of large numbers of (sometimes seemingly unimportant) details. On the other hand, system initiative is inadequate for total control because the system doesn't always know what the programmer should be doing. The idea of a *mixed initiative* approach is a good compromise, where some actions are user initiated and some are system initiated.

The DA will provide a mixed initiative form of interaction. The user initiative part is fairly obvious: the user can ask to create, modify, search, and view documentation. The system initiative aspect is somewhat unique, and provides the DA with a capability not generally found in programming environments. Based on the model for determining when documentation needs to be created/updated, and based on the context model of what the programmer is doing and what state the system is in, the DA can take the initiative to ask the programmer to update documentation at an "appropriate" time.

Having the DA prompt the user for documentation is a significant step. It means that remembering to find and update (or create) appropriate documentation is no longer strictly the programmer's responsibility. It means that the programmer does not have to switch contexts in order to update documentation. It

means that the programmer does not even have to be concerned with using the correct tools for performing the update. The act of updating documentation is considered an integral part of the editing environment; moreover, keeping track of the state of all the documentation becomes the responsibility of the system and not the programmer.

4.3 TRAVERSAL

The term *traversal* refers to moving through the documentation. There are two basic modes of traversal: *browsing* and *navigating*. Browsing, or undirected traversal, is the process of going through the documentation without looking for anything in particular. For example, if you were given a program you had never seen before and told to find the bug in the program, the first thing you might do would be to browse through the documentation just to see what it looks like. Browsing is characterized by wandering or undirected search; if you were to look over a browser's shoulder, it would be difficult to figure out what he was looking for. Browsing is unstructured: the user neither needs nor wants restraints.

On the other hand, navigating, or directed traversal, is a more goal-oriented search. For example, if you were given a program you were familiar with and told to find a bug in the program, you might have a very good idea of where to start looking.

Navigating is a more structured process, and providing support to help structure and track the search space is natural. For example, there are two basic strategies for navigating hierarchies: breadth-first and depth-first.[1] To provide support for a user operating in either of these modes, a system can keep track of where the user has been and which place should be visited next. Thus, when a level is popped, the user will find himself where he left off, without keeping mental or written notes. Since it is often the case that a person will use a cross between these two strategies, it is important to provide a mechanism that is flexible enough to provide both alternatives.

There is also another technique that is useful during navigation and possibly during browsing. This is the idea of *guided trails*. If you were maintaining a program, and knew quite well how that program worked, you might want to somehow encapsulate that information so that others could learn without repeating all your efforts. Imagine that you were to explain a certain aspect of the program to someone. You would start out in a certain place, point out relevant things, move to another place, point out other things, etc. The path followed as you move through the code is a guided trail; you have figured out what parts of the system are necessary to examine in order to understand the operation of some aspect. If there is a way of saving this trail, then it could be "played back" later (to others, or even to yourself, since you might have forgotten the trail).

---

1 In a study on program comprehension, we found that these strategies were an important characteristic of the process of debugging programs [Domeshek-84].

4.4 RETRIEVAL

Documentation retrieval is similar in many ways to navigation: searching through the documentation space for particular information. The difference is that in navigation, each move may be charted on the basis of the previous move; in retrieval, it is known how to get directly to the information.

The standard technique for retrieving documentation (or, for that matter, most any kind of on-line information) is via information/database retrieval languages. The IPE will provide a retrieval language called the Program Reference Language (PRL); while this language is aimed at retrieving programs, it could also be used to retrieve other kinds of objects. The PRL may be well suited to documentation retrieval because it is designed for retrieval of structured objects.

Another technique for retrieving documentation from the IPE is fairly obvious: *pointing*. If an object is on the screen, retrieval can be done by pointing to the object (with a cursor or a mouse) and asking for all the documentation associated with that object. Retrieval is done simply by traversing the links from the program database to the documentation database.

4.5 FORMATTING

While the DA is primarily concerned with on-line documentation, it needs to be capable of converting documentation into a hardcopy form. Conversion of documents that have a "book-like" form (e.g., manuals, specifications, plans) into documents is fairly straightforward; it is only necessary to provide output that is compatible with a text formatter. On the other hand, conversion of in-line comments into hardcopy form requires much more work. Since there may not be any explicit linear structuring in this documentation, the DA will provide a means for selecting and composing these objects.

5. FEASIBILITY

To assess the feasibility of the DA, we examine here two major issues: feasibility of building the system ("implementation") and feasibility of placing the system in use ("deployment").

5.1 IMPLEMENTATION FEASIBILITY

The DA will be rather eclectic: some sections of the DA represent new code that must be written specially for this application; some sections represent experimental code borrowed from other research efforts; and other sections may be commercially available software. These various pieces of software are discussed below.

5.1.1 The Intelligent Program Editor

Many of the ideas for the DA have come out of the IPE effort at AI&DS; in fact, the DA will actually be built upon/into the IPE system. With this bootstrapping, much of the start-up effort that would have been required for the DA will not be necessary. The IPE will provide part of the database (the EPM) as well as a user interface.

However, it should be noted that the IPE is a research effort. The IPE is still in the design phase, and it will be some time before any of the IPE system could be used to support the DA. Moreover, while one of the goals of the IPE effort is to produce a runnable prototype, there are no assurances that the IPE effort will actually produce a system that provides a usable base; as a research effort, the goals of the project are to demonstrate concept feasibility, not to deliver usable software.

The DA could be implemented without the Extended Program Model of the IPE. However, the DA would then forfeit the ability to link code directly to documentation (and hence the ability to automatically detect outdated documentation). The DA would still have a structured documentation database (though somewhat less sophisticated), tools for manipulating that database, a policy model for controlling documentation, and a user preference model for conforming to user desires.

The DA could also be implemented without the IPE user interface. However, unless additional effort were put into the development of a DA user interface, much of the interactive, integrated, window-oriented nature planned for the DA would be lost. The DA would still have much of its original functionality; it would just not be as easy to use.

Thus, the best approach to building the DA would be to utilize as much of the IPE as is available. Building on the IPE can only help the DA; its unavailability would impact only the time necessary to build the DA.

5.1.2 Documentation Database

The documentation database for the DA will be developed in two steps. The first step is to develop a stand-alone mode: a database for documentation that provides support for just documentation and is not linked to the Extended Program Model. The second step is to integrate this database into the EPM, so that documentation and programs can be linked together.

Developing a stand-alone documentation database is definitely feasible. The best approach would be to develop a special purpose database in order to address intricacies specific to documentation. As a backup approach, an existing database system could be used, and necessary database routines could be added.

Integrating the documentation database into the EPM is more of an unknown. Since the EPM is still under design, determining the ease of adding a new component is difficult. However, the EPM is being designed with the intention of being able to add documentation. Thus, it is likely that adding the documentation database to the EPM will be significantly less work than the development of the EPM itself.

5.1.3 Detection of Outdated Documentation

One of the most unique features of the DA is its ability to automatically determine when documentation is out of date. There is a great deal of variability possible in the effectiveness of this task. To better describe this process, it is useful to borrow a few terms from the field of information retrieval. In the context of the DA, the term *recall* refers to the percentage of outdated documents that were detected; the term *precision* refers to the percentage of detected documents that were actually outdated documents.

The goal of the DA (and of any retrieval system) is to achieve maximum recall and maximum precision. Unfortunately, it is often the case that strategies for trying to increase one measure results in decreasing the other. Therefore, any strategy aimed at increasing one measure needs to be carefully monitored to determine the effect on the other.

The simplest strategy for detecting outdated documentation is to flag documentation as outdated if anything it references is changed. This is likely to result in high recall but very low precision; along with the documentation that actually needs changing, the system will flood the user with documentation that is not outdated. Based on current technology, this is about the best that can be done. The DA should be able to surpass this, achieving a similar degree of recall but with a significantly higher degree of precision.

The most prudent approach to increasing precision appears to be the incremental addition of knowledge about the documentation taxonomy (provided by the DA itself) and knowledge about the program semantics (provided through the EPM). The utilization of this knowledge makes it possible to eliminate more

pieces of documentation from the list of possibly outdated documentation. The purpose of incremental addition of knowledge is to add knowledge in small chunks, making it easier to determine the kinds of knowledge have a negative side effect (in terms of precision and recall).

By using these techniques, it is quite likely that the DA can achieve its goal of higher recall and precision. Just what levels can be achieved is an open question, as is the question of what levels users will find acceptable.

5.1.4 Documentation Retrieval

The documentation retrieval facilities discussed earlier should be relatively easy to build. In the case that users find themselves in need of a more general purpose retrieval capability (e.g., unformatted text retrieval), it may be possible to make use of an existing information retrieval system. Commercially available information retrieval systems provide basic capabilities for locating documentation. Better yet, AI&DS has already developed a prototype of an intelligent information retrieval system that provides retrieval capabilities beyond that of any commercially available system [McCune-83]. Incorporating one of these retrieval systems should be a straightforward effort.

5.1.5 Documentation Formatting and Analysis

The DA effort will not require the development of any new documentation formatting tools; rather, it will rely on existing document/text formatters. The only difficult part of document formatting is linearizing the database/network of documentation, coercing it into a form acceptable by the text formatter. Given that the documentation database (and the tools for manipulating it) are available, there should be little difficulty in achieving this. Similarly, existing tools for checking spelling, grammar, diction, readability, etc. could easily be added to the system.

5.2 DEPLOYMENT FEASIBILITY

Assuming that the DA can be successfully built, the next question is the feasibility of moving such a system out of the research environment and into a production environment. The following subsections provide some insight into these issues.

5.2.1 Current Documentation Problems

The need for a tool like the DA is great, especially in government/military programming environments. Current documentation practices in these environments suffer from many problems. A number of these problems were identified in an earlier study of software maintenance [Dean-83], which found that documentation takes a back seat to software, for a variety of reasons:

1. Documentation is not considered an important part of the end product:
    - only a portion of the total documentation is specified as a deliverable
    - the requirements for deliverable documentation are much less rigid than the corresponding requirements for the software
    - documentation is often not done until after the programming
2. Documentation is not considered an important part of the software life cycle:
    - documentation is allowed to lag behind the software
    - programmers dislike writing documentation
    - documentation writers have inadequate training
    - writers are presented with little structure and inadequate guidelines
3. Documentation is poorly handled:
    - tools for manipulating documentation are inadequate
    - documentation is sometimes done off-line
    - it is difficult to evaluate the completeness or correctness of documentation

The DA provides a basis for helping reduce these problems. The first problem would be partially alleviated if contractors used the DA during the program development process, since their documentation task would be eased, and their coding environment would be integrated with their documentation environment. The second problem is also partially addressed by the DA; while no tool can make someone write documentation, a good tool can encourage and assist the process. The DA certainly forces the recognition that documentation is a critical part of the software life cycle. The last problem is directly addressed by the capabilities of the DA.

5.2.2 Documentation Life Cycle Support

The DA, as described in this document, represents only one viewpoint of a documentation system (that of the programmer). For any documentation system to be truly usable, it must provide facilities for all those involved in the production and maintenance of documentation, including designers, managers, technical writers, typists, and testers. Each of these user categories has its own unique needs. For example, a designer might want support for the development of

documentation in the form of a program design language; a project manager might want tools to do consistency checking on documentation, to make sure that the documentation corresponds to the code.

The DA is being designed to accommodate the needs of all these users. The basic underlying mechanisms (i.e., database, user interface) will be in place; all that will be required is the development of additional tools, built on the existing system, to support these different needs. Thus, the proposed version of the DA represents one facet of a documentation support system (see Figure 5.1). It will eventually be capable of supporting the entire documentation life cycle, but this will require further development.

5.2.3 Supporting Documentation Standards

The DA is designed to support different documentation standards. To give a better picture of how this might be done, this section presents a portion of the proposed SDS military standard [SDS-83]. In the example presented here, just one path of the SDS documentation tree is traversed, and an example of how the DA might represent this information is shown. This example is meant to provide a rough sketch; it should not be construed as a complete picture of SDS nor as a complete picture of documentation representation techniques.

Figure 5.2 serves as a map to this discussion; it shows the path through the SDS documentation that will be traversed in this example. As presented here, the documentation looks hierarchically arranged, but this is a simplification made for the purpose of presentation. At each level, the item in boldface is the item that will be focused on at the next level.

At the top level of the hierarchy (Figure 5.3) are two items, DOD-STD-SDS, the proposed military standard on defense system software development, and a set of Data Item Descriptions (DIDs). The SDS standard references all of the DIDs.

The Data Item Descriptions describe the documents that contractors must deliver with software. There are approximately 25 DIDs referenced by SDS. Each DID may reference a number of other DIDs. Figure 5.4 shows the overlap between DIDs. The figure also distinguishes between several different categories of DIDs: requirements/design documents, testing documents, and user documents.

Each DID describes the structure of a document. Figure 5.5 presents the structure in the Software Test Plan (STP). From Figure 5.4, it can be seen that the STP references several other documents. These references are shown in more detail in Figure 5.5, where the arrows represent specific places in the STP that reference other documents.

In addition to the document structure, each documentation object has attributes associated with it, as discussed earlier. In Figure 5.6, there is an example of some of the attributes that might be associated with the Software Test Plan DID.

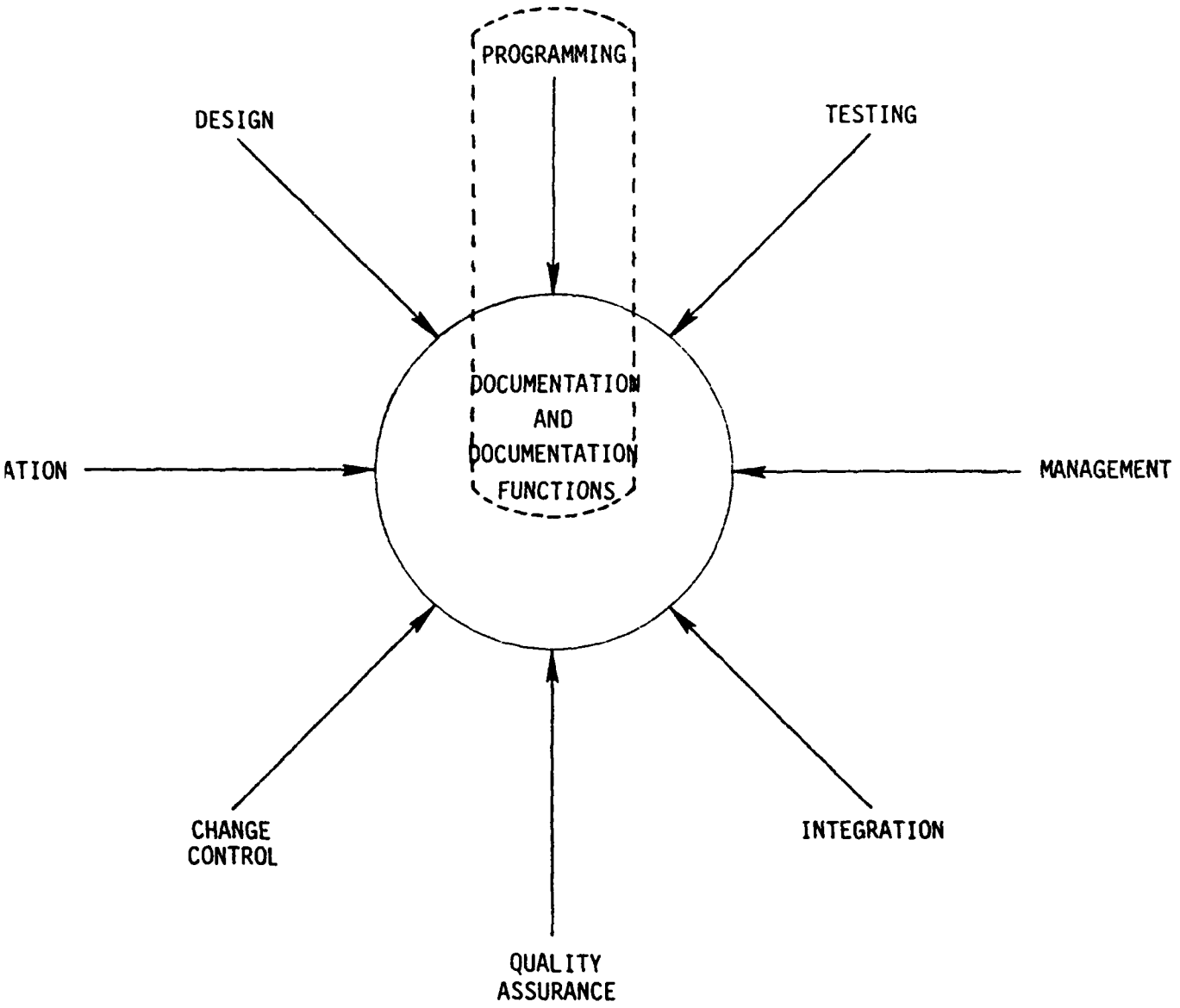


Figure 5-1: Documentation Life Cycle Support

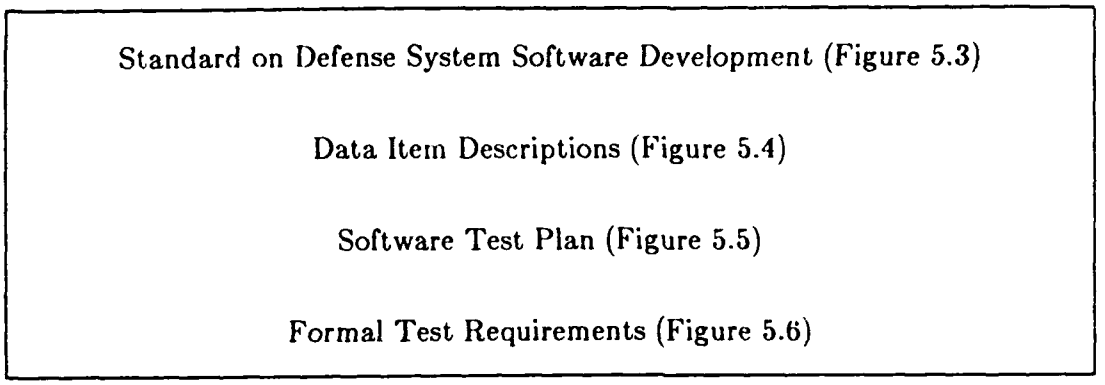


Figure 5-2: Overview of SDS Documentation

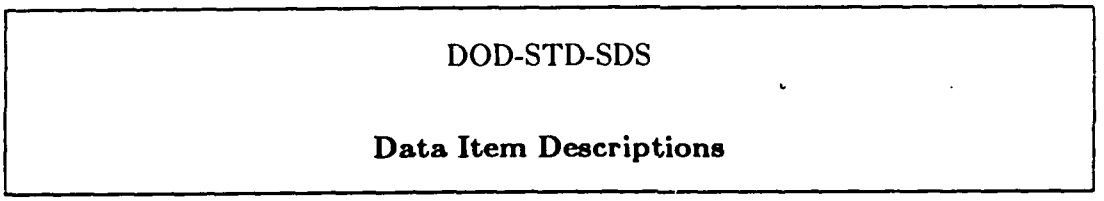


Figure 5-3: The SDS Standard (top level)

There may be additional substructure to the structure presented in Figure 5.5. Figure 5.7 presents an example of how to fill in the section entitled "Formal Test Requirements." Unlike the higher level structures, this structure presents suggestions to the author of the document, rather than stating necessities.

The DA weaves these levels together into a network that represents the relationships and interconnections. Figure 5.8 is an example of how this network might be represented by the DA. Objects on the left hand side of the figure represent classes of objects; objects on the right hand side represent instances or subclasses of these classes. For example, DOD-STD-SDS is an instance of a standard; the Software Test Plan is a subclass of Data Item Descriptions.

SDS is but one example of what documentation structure might look like; certainly, many other structurings are possible. As indicated by this example, the structuring mechanisms provided by the DA are meant to be quite general in order to provide support for any type of documentation structure.

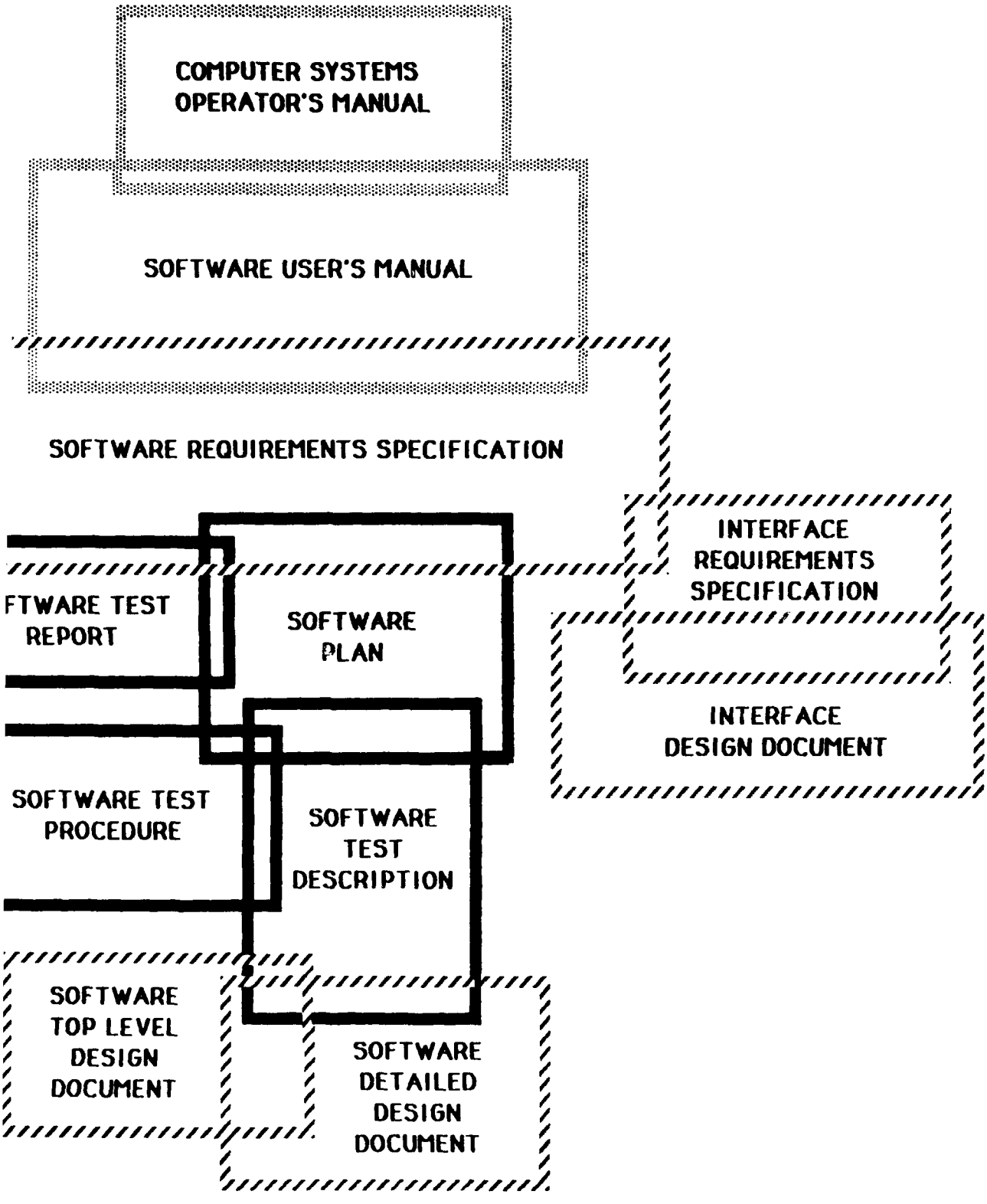


Figure 5-4: Data Item Descriptions (partial list)

APPENDIX A

A Knowledge Base for Supporting an Intelligent Program Editor

9. REFERENCES

[Dean-83] Dean, J. and B. P. McCune, "An Informal Study of Software Maintenance Problems," Software Maintenance Workshop, Montery, California, December 1983.

[Domeshek-84] Domeshek, E., D. Shapiro, J. Dean, and B. McCune, "An Informal Study of Program Comprehension," AI&DS TM-1014-3, March 1984.

[McCune-83] McCune, B., R. Tong, J. Dean, and D. Shapiro, "RUBRIC: A System for Rule-Based Information Retrieval," Proceedings, COMPSAC '83, IEEE Computer Society, pp. 166-172.

[SDS-83] Proposed Military Standard on Defense System Software Development, DOD-STD-SDS, December 1983.

[Shapiro-84] Shapiro, D., J. Dean, and B. McCune, "A Knowledge Base for Supporting an Intelligent Program Editor," Proceedings, 7th International Conference on Software Engineering, March 1984, pp. 381-386.

8. CONCLUSION

Despite the tremendous need for support of the documentation process, there has been little research in this area aimed at making significant improvements to the current techniques used. As an intelligent tool designed to assist users in all phases of the documentation life cycle, the Documentation Assistant represents a significant step in this direction.

The focus in this report has been on documentation from the programming viewpoint; this is a logical place to start, given that programmers already use various computer-based tools on a regular basis. In addition, current research on programming environments is leading towards tools like the Intelligent Program Editor (under development at AI&DS), which provide a natural base for building documentation tools.

However, to reach the goal of providing total documentation life cycle support, it is clearly necessary to address the needs of the many other users of documentation. This report should be viewed as a first step in that direction. The DA provides a framework for manipulating documentation that should easily extend to provide this support. To provide support for the entire life cycle, additional tools and techniques must be developed.

To construct the DA, it will be necessary to make use of existing technology, experimental technology, and brand new technology. The building of a system like the DA definitely has risks associated with it, especially the parts of the system that require new technology to be developed. However, given the mix of technologies that will be used by the system, it is likely that a significant portion of a DA prototype can be built with minimal risk.

With the rapidly escalating growth (and cost) of software maintenance, there is a clear need for the development of new tools and techniques to handle the problems and bottlenecks associated with the software process. Of these issues, documentation is possibly the biggest and most crucial one. The Documentation Assistant is aimed precisely in this direction.

documentation that is not in the database. Tools and techniques for aiding the conversion of this documentation may eventually need to be developed.

- *Programming Context Model:* The purpose of the Programming Context Model is to keep track of what the programmer is doing. However, current techniques for doing this are overly simplistic, and if accuracy is required, a good deal of user cooperation is needed. The development of more sophisticated techniques for determining what the user is doing would ease the task of the user, as well as increase the reliability of this component of the system.

- *Improved Detection of Outdated Documentation:* By making use of the semantic information that the EPM will provide, it is possible to increase the accuracy of determining outdated documentation. There is a great deal of information to be gained from the various semantic representations; just how much accuracy can be achieved is an open question.

- *Consistency Maintenance:* The issue of consistency maintenance is a critical one for both the IPE and the DA. Consistency checking can be done at many levels. For example, when a documentation object changes, it is necessary to track down any other dependent documentation objects. However, determining these dependencies can be difficult. Even if the dependent documentation explicitly references the changed documentation, it is unclear if the change actually affects the dependent object; even worse, the dependent object might reference the changed documentation implicitly. As the DA grows more sophisticated, it is necessary for the consistency maintenance mechanisms to follow.

- *Knowledge Acquisition:* A great deal of information/knowledge is necessary for the optimal functioning of the DA. To port the DA to different environments (or to modify what is known about current environments) old knowledge must be modified and new knowledge must be added. Tools for aiding this knowledge acquisition process are essential, especially if the task is to be performed by someone other than the system developers. For example, in the case of rule-based systems, knowledge acquisition tools include structured rule editors (to insure that only syntactically correct rules are entered), rule evaluators (for determining the effect of rule sets), rule analyzers (for determining properties such as sensitivity, connectivity, consistency), etc. Similar tools will be needed for the DA.

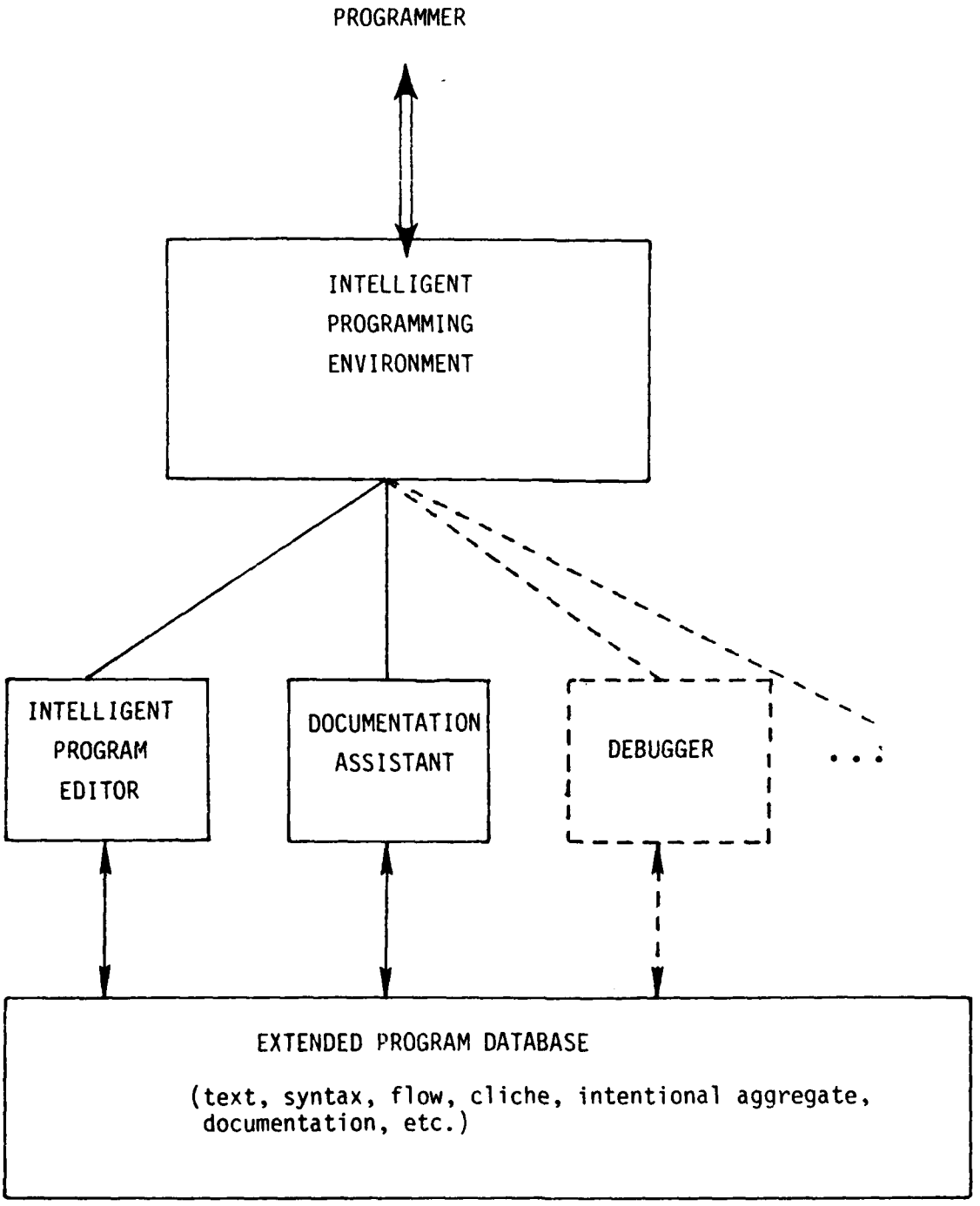


Figure 7-1: An Architecture For Advanced Programming Environments

7. FUTURE RESEARCH

The plan for building a prototype of the DA necessarily omits many of the issues and areas requiring significant research efforts. Some of the research issues that might be addressed in the future are discussed below.

7.1 FUTURE RESEARCH ON PROGRAMMING ENVIRONMENTS

The IPE and the DA are two important components of an advanced programming environment currently being developed at AI&DS. The architectural basis for this environment is the intelligent program/documentation database capability provided by the Extended Program Model, which will provide functionality usable by a variety of tools. While the combined IPE/DA system would provide a great boost in capability over existing programming environments, there is still at least one important aspect of programming environments that our research does not currently address.

To increase the usefulness of the IPE/DA, the next step would be to incorporate support for the dynamic aspects of programming (i.e., tools to support program execution). These tools would also be based on the EPM (Figure 7.1), and thus would benefit by having access to the multiple representations provided by the EPM, including the linkages connecting various program segments and documentation. With access to this additional information, these tools could provide capabilities beyond those provided by current runtime support environments.

7.2 FUTURE RESEARCH ON DOCUMENTATION

With respect to documentation, there are a number of directions that would be logical for further advanced study. A brief discussion of these issues follows.

- *Documentation Life Cycle Support:* Support for all people involved with the production and maintenance of documentation must be provided. This includes support tools for different documentation types (e.g., requirements, specification), document entry (e.g., checking spelling, grammar, style), document monitoring and control, and consistency checking. The database provided by the DA will be an ideal base for the development of these tools, since it provides documentation already in a highly structured form suitable for machine analysis.

- *Retrofitting to Existing Systems:* The discussion so far has assumed that all documentation is already in the documentation database. For new systems that are built using the IPE/DA, this might be a reasonable assumption, since documentation will be entered into the database as soon as it is entered into the computer. However, for programs that have been developed without the aid of these tools, there will be

moderate, primarily because the IPE must be modified to accommodate documentation. However, the highest risk is assuming the availability of a workable version of the IPE. Even if a prototype of the IPE is available, it is unclear how easy it will be to modify the system.

Year 5 (1989):

The fifth year of the project will result in a more fully integrated version of the IPE/DA, allowing a user to easily move between programs and documentation. Advanced capabilities will begin to be developed, including the ability to model user preferences about various states of the system, as well as the ability to automatically detect outdated documentation.

As in the previous year, if the IPE system is not sufficiently developed, integration of the DA into the IPE will be hindered. The task of providing advanced capabilities for user modelling and outdated documentation detection should be considered research topics, and thus may be fairly high risk.

The integrated version of the DA will be demonstrated at the end of this year. An evaluation of the system will also be performed, by applying the DA to some subset of the code and documentation for a real software system (to be selected at some future time).

6.2 TASK DESCRIPTIONS

The following paragraphs describe the tasks in greater detail and describe the risk associated with each set of tasks.

Year 1 (1985):

The major focus of the first year of effort will be the design of the DA system. A large part of the effort will be to examine currently existing databases and database tools to discover current technology that could be used in the system itself or as an aid during system development. During this year, the internal representation for the documentation will be developed. The design of the user interface will be undertaken and will make use of the Rapid Interface Prototyper system that is currently under development at AI&DS. This system allows for the user interface designer to study different styles of user interface display and actions before actually committing the ideas to code.

Years 2 & 3 (1986/1987):

The second and third years of effort will be concerned with the construction of the actual database, making use of existing tools and technology whenever possible. The database manipulation functions will be developed during this time. To enable incorporating existing documentation into the DA, methods will be developed to allow the user to interact with the system to add preexisting text to the database. Finally, a primitive "word processing" mode will be added to the user interface to ease the process of entering larger amounts of text into the system.

Depending on the availability of existing software, the risk involved in the construction of the documentation database may vary; it would range from low risk (in the case that an existing database system could be used) to moderate risk (in the case that the database must be designed from scratch).

This initial version of the DA will be used for internal experimentation. At the end of this period, a demonstration of this system will be provided.

Year 4 (1988):

Much of the fourth year of effort will be spent in the integration of the DA into the IPE. As a major step in this task, the ability to associate documentation with code will be added. This will allow the user to point at a piece of code, request to see the documentation associated with that code, and have the text automatically displayed. In a similar fashion, it will be easy to add code-level documentation to the database at the same time the code is being written. This year will also include work on providing documentation formatting functions that will allow hardcopy documents to be produced from the documentation database.

For tasks not involving the IPE, the risk for this year is low; it mainly involves providing different ways of manipulating documentation. Since a large portion of the effort for this year will build on the IPE, the risk is compounded. Assuming that the IPE prototype is available, the risk for the remaining tasks is

6. WORK PLAN

This section presents a plan for the design and implementation of an exploratory development prototype of the DA. This prototype will be able to make use of the technology and tools being developed as part of the Intelligent Program Editor Project, another Navy research project currently in progress at AI&DS.

6.1 TASK SUMMARY

The tasks involved in building a research prototype of the DA are summarized below on a yearly basis.

Year 1 (1985) [estimated effort: 0.8 person/years]

- Develop an overall system design
- Design the documentation representation formalism
- Design and prototype user interface

Years 2 & 3 (1986/1987) [estimated effort: 1.0 person/years per year]

- Construct database
- Design and implement database functions
- Implement a "word processing" mode for documentation
- Demonstrate initial prototype (end of 1987)

Year 4 (1988) [estimated effort: 1.5 person/years]

- Design needed functionality to integrate into IPE
- Associate documentation with code
- Provide documentation formatting capability

Year 5 (1989) [estimated effort: 2-5 person/years]

- Complete integration into IPE, with ability to document different views
- Basic method for automatically detecting outdated documentation
- Basic user modeling capability
- Demonstrate and evaluate integrated prototype (end of 1989)

It will also be important for knowledge-based tools to provide functionality even in applications where only a minimal amount of knowledge has been collected. Of course, there is a real tradeoff here: as knowledge is reduced, the ability to act intelligently is also reduced. It makes little sense to employ intelligent tools without any of the knowledge needed to act intelligently (one might say that this is a non-intelligent application of technology). While the DA is dependent on a documentation knowledge base, it does provide a number of useful capabilities requiring only a small amount of knowledge engineering. These features include the documentation database, documentation interconnection (though all interconnections must be explicitly specified by the user), and tracking (though the user must manually enter the appropriate information).

These tradeoffs can be described as striking a balance between the work required by the user and the work required by the system. As the knowledge base grows (along with the ability of the system to handle that knowledge base), the amount of work required by the user decreases. Cutting back on the knowledge base does not eliminate the possibility of using an intelligent system; it just shifts the burden for much of the legwork from the computer to the user.

The issue of who performs knowledge engineering (and its subsequent maintenance) is one that the research community has still not adequately answered. To date, the people who have done knowledge engineering have been AI researchers (or people trained by them). The issue is particularly important in production environments, where continual environmental changes necessitate corresponding changes to appropriate knowledge bases. For intelligent systems to achieve wider usage, it is necessary to develop tools and techniques for allowing people who are not AI experts to perform knowledge engineering and maintenance tasks.

5.3 FEASIBILITY SUMMARY

The more "intelligent" aspects of the DA will be based primarily on research already in progress at AI&DS (as part of the ONR-funded Intelligent Program Editor project). Existing software tools that provide other capabilities (e.g., text formatting, information retrieval) could be incorporated into the DA system.

Many current documentation problems faced in real programming environments are addressed, either directly or indirectly, by the DA. The primary cost associated with using the DA (apart from development costs) is the initial collection and codification of the documentation knowledge base. This is a one-time cost for each programming project/environment; however, much of this information should be reusable, especially between similar environments.

5.2.4 Knowledge Acquisition and Maintenance

The difference between the current generation of programming systems and future generations (e.g., the so-called *Fifth Generation* systems) is knowledge. Current generation systems have very little knowledge about how they are being used and about the semantics of their intended application. For example, programmers today usually use text editors to edit programs. There is a great deal of syntactic and semantic information that a program editor could use; ignoring this information reduces the capabilities of the system and fails to reflect the conceptual levels at which programmers work. Yet this increased capability is precisely what is required to increase the usefulness and productivity of computer systems.

There is, however, a price to be paid for making systems more intelligent, and that is the cost of knowledge. Knowledge is a rather expensive commodity. It is costly to hire an expert to help apply knowledge; it is costly to codify knowledge; it will be costly to take that knowledge and incorporate it into software that exhibits intelligence.

The artificial intelligence community is quite aware of the value (and cost) of knowledge. They have coined the term *knowledge engineering* to describe the process of acquiring and codifying knowledge. Most AI systems are restricted to narrow domains in order to reduce the amount of knowledge necessary, thus reducing acquisition costs as well as reducing the size of the search space.

Despite the cost of knowledge, intelligent systems can indeed be worth the expense. The gains in productivity and effectiveness provided by intelligence should more than compensate for the increased costs. However, it should be recognized from the beginning that it will cost more to develop intelligent systems; the payoffs come later as systems are put to actual use, and the intelligent systems prove more economical than their "dumb" counterparts.

There are two basic approaches to reducing the costs and risks associated with intelligent systems development. First, as knowledge based systems become more common, it may be possible to reuse existing knowledge, at a much lower cost than redoing the entire knowledge engineering process. It will also be possible to share knowledge among different instances of a particular system. Second, as a means for evolving gradually towards more intelligent systems, intelligent tools can be usable even with minimal knowledge bases. By reducing the initial knowledge engineering effort, costs can be controlled until the concepts have been proven.

The knowledge in the DA takes many forms; for example, there is knowledge about the structure of documentation, knowledge about the documentation process, knowledge about what the user is doing, knowledge about policies and user preferences, and knowledge about interacting with the user. Some of this knowledge is environment or site specific, and some of it will be valid for many sites. The cost of reusing knowledge will be less than the cost of regenerating it (just as the cost of reusing software is less than the cost of rewriting it).

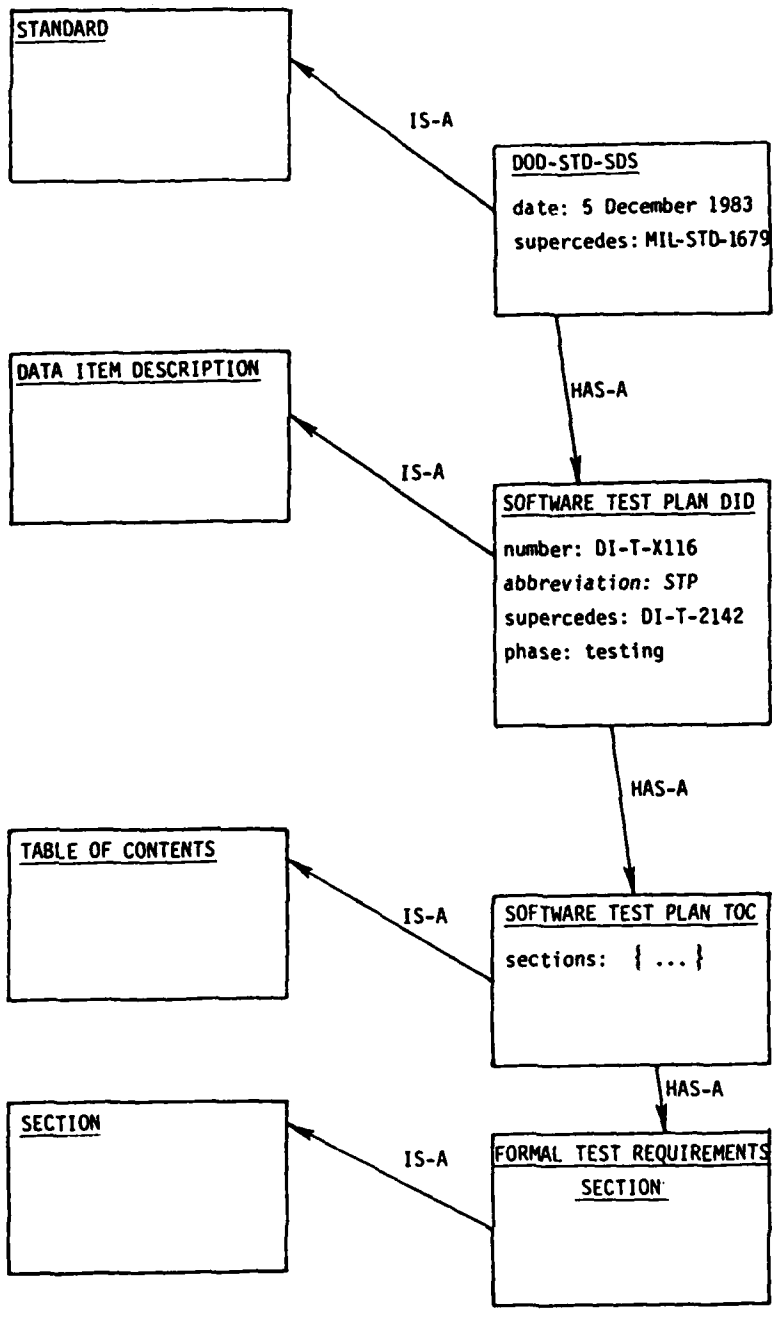


Figure 5-8: Representation of the SDS Documentation Hierarchy

Software Test Plan	
attribute	*value*
Number	DI-T-X116
Abbreviation	STP
Predecessor	DI-T-2142
Agency	Navy
Phase	testing
Scope	single CSCI
Length	13 pages
Date	5 December 1983

Figure 5-6: Attributes of the *Software Test Plan* DID

All formal tests shall include the following test requirements:

- The size and execution shall be measured.
- Nominal, maximum, and erroneous input and output values.
- Error detection and proper error recovery, including appropriate error messages.

Formal Tests for radar tracking requirements shall include the following test requirements:

- Simulated test data on all possible combinations of environmental conditions.
- Input data taken from the environment ("live data").

Figure 5-7: Example of *Formal Test Requirements*

Limitations/Traceability
- Limitations
- Traceability → *Software Requirements Specification*, *Interface Requirements Specification*

Informal Test Plans
- Unit Testing
  - Unit Test Requirements
  - Unit Test Management
  - Unit Test Schedule
- CSC Integration Testing
  - CSC Integration Test Requirements
  - CSC Integration Test Management
  - CSC Integration Test Classes
  - CSC Integration Tests
    - ....
  - CSC Integration Test Table → *Software Requirements Specification*
  - CSC Integration Test Schedule
- Resources Required
  - Facilities
  - Hardware
  - Interfacing/Support Software
  - Source

Formal Test Plans
- **Formal Test Requirements**
- Formal Test Management
- Formal Test Classes
- Formal Tests
  - ....
- Data Reduction and Analysis
- Formal Test Table → *Software Requirements Specification*
- Formal Test Schedule
- Formal Test Reports → *Software Test Report*
- Resources Required
  - Facilities
  - Hardware
  - Interfacing/Support Software
  - Source

Notes
Appendix

Figure 5-5: Software Test Plan DID

[Proceedings, 7th International Conference on Software Engineering, Orlando, Florida, March 1984]

A KNOWLEDGE BASE FOR SUPPORTING AN INTELLIGENT PROGRAM EDITOR

Daniel G. Shapiro
Jeffrey S. Dean
Brian P. McCune

Advanced Information & Decision Systems
201 San Antonio Circle
Mountain View, CA 94040

ABSTRACT

This paper presents work in progress towards a program development and maintenance aid called the Intelligent Program Editor (IPE), which applies artificial intelligence techniques to the task of manipulating and analyzing programs. The IPE is a knowledge based tool: it gains its power by explicitly representing textual, syntactic, and many of the semantic (meaning related) and pragmatic (application oriented) structures in programs. To demonstrate this approach, we implement a subset of this knowledge base, and a search mechanism called the Program Reference Language (PRL), which is able to locate portions of programs based on a description provided by a user.

This research was supported by the Air Force Office of Scientific Research under contract F49620-81-C-0067, the Office of Naval Research under contract N00014-82-C-0119, and Rome Air Development Center under contract F30602-80-C-0176.

1. INTRODUCTION

The effort and expense involved in software maintenance have been recognized as a major limitation on the capabilities of current software systems. In a study on software maintenance issues in the Air Force, we found that the process of comprehending the form and function of existing software (i.e., what it does and how it does it) is the largest task in the maintenance process [2].

The basic cause of this "comprehension problem" is the loss of knowledge during the programming process, caused by factors such as poorly written software, inadequate documentation, programmer forgetfulness, and personnel turnover. To address these issues, we have started a project to develop intelligent, knowledge-based programming aids, designed to help the programmer overcome limitations of more traditional tools. This paper describes the initial phase of one of these tools, an editor known as the Intelligent Program Editor (IPE). The following sections discuss the motivation behind intelligent editing, the design of an intelligent editor, a database for the editor, and a scenario demonstrating an actual implementation of a portion of the IPE's database, used in the context of a program search.

2. MOTIVATION

An intelligent editing system is a sophisticated tool for developing and maintaining programs. The goal, insofar as it is possible, is to decrease the amount of information a programmer needs to supply in order to create and maintain a program, and to simultaneously increase the reliability of the resulting code. This can be accomplished by incorporating knowledge about the structure and intention of programs into the editing tools used to develop and maintain them. Perhaps the best way to illustrate this approach is to present an allegory having to do with the production of a technical manuscript.

Assume that there is a manuscript which needs to be typed for publication. If it is given to a typist who does not speak English, the result would be, at best, a word-for-word copy of the original manuscript. If it is given to an English-speaking typist, simple errors, such as misspellings and punctuation problems, might be fixed during the typing process. If the manuscript is given to an English teacher moonlighting as a typist, the result might well be a version in which the prose is smoothed and otherwise improved. Finally, if one is lucky enough to find a typist familiar with the domain of discourse (such as the author), the resulting document might even have factual errors corrected and incomplete thoughts identified.

A programmer selecting an editor system for writing code is in a similar situation. A standard text editor is comparable to the non-English-speaking typist; text appears exactly as it is typed, with no enhancements. The English-speaking typist could be compared to a syntax-oriented editor, which can eliminate syntactic program errors and misspelled keywords. The English teacher/typist knows about the language itself but not about the content of the thoughts. This situation is comparable to a programming language-specific editor which applies knowledge about the domain of programming; this editor can instantiate general programming techniques, catch certain types of semantic errors, make style suggestions, and improve the overall flow of the program. The technical typist who understands the content of what is being said is analogous to an editor that

utilizes knowledge about the application domain; it can help in algorithm development and can catch certain types of pragmatic errors which are dependent upon the specific application domain.

3. THE INTELLIGENT PROGRAM EDITOR

The Intelligent Program Editor (IPE) described in this paper most closely corresponds to the English teacher/typist mentioned above, in that it will support textual and syntactic manipulations, and have the ability to assist in the implementation of typical programming actions. This power is obtained through the use of a database that explicitly represents the functional organization of programs in terms of textual, syntactic, and intention-oriented structures. With this database, the IPE is in a position to become more of a programming environment than solely an editing tool. In this vein, we are interested in addressing the following design goals [5].

The IPE should provide a means for naturally incorporating documentation into the program development process. In our view, this requires the ability to link documentation into the organizational structure of a program (similar to Nelson's [3] concept of Hypertext), and the ability to actively use any information that is supplied (to provide programmers with a motivation for including descriptive data). In the IPE, documentation will provide input to a program search facility.

The system should support incremental program analysis. The object here is to employ the system's understanding of program structure to catch syntactic and certain semantic errors prior to execution. Examples include identifying variables that are accessed before being set (via data flow analysis) and detecting programming cliches that have been incompletely implemented. There is also a role for error prevention: some editors (e.g., [6]) prevent syntactic errors from ever occurring.

The IPE will allow the user to employ alternate program visualizations. This means allowing the programmer to examine or modify code through any of the representations mentioned above. For example, a syntax based approach might be appropriate during program construction, while a graphical data flow display may be useful within the debugging process.

All of these capabilities require the use of multiple program representations, as well as mechanisms for searching and manipulating the information they contain. Therefore, in the first phase of the IPE project, we constructed a prototype version of this program database, called the Extended Program Model (EPM), and demonstrated it in the context of program search. The remainder of this paper discusses the EPM and the search example that was produced.

4. THE EXTENDED PROGRAM MODEL

The Extended Program Model (EPM) provides a new way of representing and accessing programs by defining a vocabulary for discussing programs which uses terms that are much closer to the ones which users naturally employ. The EPM provides this capability through the use of a database that represents the structure of programs from a variety of views. The EPM can form the backbone for a number of systems which exhibit a deep understanding of the organizational structure and meaning of code.

The EPM is constructed in terms of two major subsystems (see Figure 1) : a program structures database and a search and update component called the Program Reference Language, which provides access to the database. In addition, the EPM will contain a library of "rational form" constraints that will monitor program composition for its structure and intentional content. As a whole, the system can be thought of as a database management system for creating and maintaining code. It provides a search language for accessing its knowledge, a facility for performing updates, as well as a set of semantic integrity and consistency constraints for monitoring the validity of the data it contains.

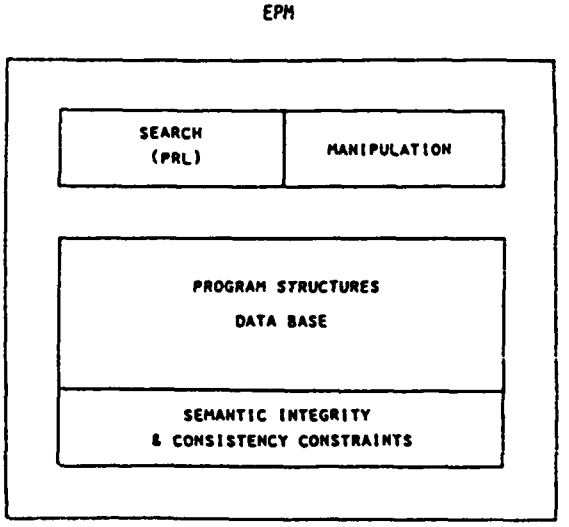


Figure 1. The Extended Program Model

4.1 THE PROGRAM STRUCTURES DATA BASE

The EPM's program structures database is constructed in terms of a collection of representations which reflect the transition from a syntactic to a more intention-oriented analysis of code (Figure 2). We are considering these viewpoints to be abstract data types which facilitate different sorts of retrieval operations.

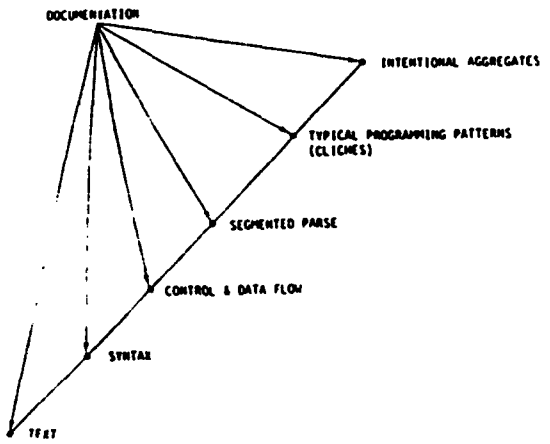


Figure 2. Representation Levels in the EPM

The textual representation gives the EPM the view that most text editors provide. It is a low-level approach, concerned with words and delimiters, but it allows for important textual search operations.

The syntactic viewpoint embodies the rules of grammar for particular programming languages. The syntactic database provides the EPM with a vocabulary for programming constructs such as "for" loops, parameters, and procedures.

The next level of representation is the flow level, which provides standard data and control flow information. It provides a vocabulary relating to the logical structure of programs.

The segmented parse representation defines a vocabulary for a program in terms of its component data and control flow. For example, iterations are decomposed into a set of roles which identify the subfunctions of a loop. In the breakdown we are using, loops contain generators, filters, terminators, and augmentations [7]. Generators are segments which produce a sequence of values. They can be further refined into initializations and a body, which is the portion that is executed many times. Filters restrict that sequence of values. A terminator is like a filter, except that it has the additional potential to stop execution of the loop. An augmentation consumes values and produces results. There are other vocabulary elements for describing straight line code.

The taxonomy discussed up to this point primarily captures information about the form of programs (as opposed to their meaning). The only semantic elements we have introduced describe the substructure of built-in entities such as loops. The next (more abstract) viewpoint considers programs to be built of objects with stereotyped purposes. These are called typical programming patterns (TPPs). Examples of TPPs include variable interchanges, list insertions, and hash table abstractions. These abstractions are the tools employed by every expert programmer. Rich has defined a library of such TPPs [4] (he uses the term cliche; in this paper, we use both terms interchangeably).

The remaining databases (intentional aggregates and documentation) provide methods for associating the intentions behind a program with specific features of code. They capture pragmatic knowledge relating to the domain of application of the program. Intentional aggregates are a type of formal documentation that allow the association of larger program fragments with key concepts (supplied by the user). They can be used to collect a set of TPPs and other program segments that implement a single conceptual function; for example, a collection of TPPs representing queue operations might be grouped (by the user) into an intentional aggregate representing a scheduler.

The documentation database allows the user to associate comments with any of the program features already described. At the lowest (i.e., textual) level, this would take the form of in-line comments. At other representational levels, the user could, for example, document the data flow in a particular module (saying why an input-output relationship occurs), justify his use of particular TPPs, or explain why particular syntactic features are employed. The advantage of this technique over current documentation practice is the ability to make a direct association (via links maintained by the IPE) between the documentation and what it talks about, at an appropriate conceptual level.

4.2 KNOWLEDGE ACQUISITION

Since the EPM's database is intended to support an actual editing system in the near future, it is important to address the question of where its information is obtained. In our approach, the different knowledge sources are acquired in part from the user, and in part by automatic means. Specifically, the syntactic representation can be obtained directly from the textual representation, and the segmented parse viewpoint can be constructed through data flow analysis techniques of the kind developed by Waters [7].

The TPP structures are harder to obtain. Recent research efforts indicate that general recognition of cliches may be possible [1], but at the current time, these techniques have not actually been demonstrated. The EPM will use manual recognition techniques (at least until automatic recognition techniques have been refined). There are two manual recognition techniques planned for the system. In the first, the user points to a piece of code and identifies it as being a particular TPP (as a way of documenting the system); at this point, once the scope has been narrowed down, it may be possible to identify the subcomponents of these programming cliches automatically. In the second method, the user uses TPPs for program generation (as in [8]); by instantiating a TPP and "filling in the blanks," the EPM can acquire all the necessary information.

The intentional aggregate and documentation views must be wholly obtained from the user. At a minimum, the EPM's planned consistency mechanisms will identify any of this information that may be out of date due to modifications to the code.

5. THE PROGRAM REFERENCE LANGUAGE

In order to demonstrate the feasibility of the EPM, we implemented a portion of the database described above, and built a version of the EPM's search facility, the Program Reference Language (PRL) which operates on that data. The PRL is a tool for locating regions of program text based upon a description provided by the user. As a support system, it provides programmers with an intention-oriented vocabulary for specifying portions of programs in situations where they may be unfamiliar with the detailed structure of the code. This might occur in the process of editing programs which may be too large to remember explicitly, or in the act of understanding code which has rarely been seen before (as is often the case in maintenance).

The PRL demonstration system allows program search based on four of the representations described above, namely the textual, syntactic, segmented parse and typical programming pattern views (Figure 3). These databases are connected through a code region abstraction that associates program features with physical sections of program text.

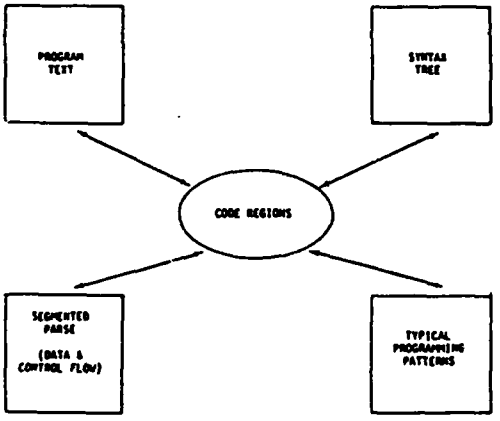


Figure 3. The Program Reference Language Implementation

The PRL has a flat information structure. It represents each database in terms of a complex tree or graph structure of frames. Although the system can arbitrarily convert between viewpoints by using code regions as an intermediary, the databases have no direct links between one another. These conversions are inherently heuristic since the separate representations do not necessarily have a one-to-one correspondence. The information in each database is either automatically derived, or can be reasonably obtained from the user. In situations where the latter is necessary, we have assumed that information may be provided in an incomplete form.

5.1 CODE PAINTING

From a computational point of view, the main problem involved with this multiple representation approach is to define a mechanism that is able to compare information obtained from the different sources of knowledge. The PRL accomplishes this via the code region abstraction, which functions as a common language that each of the representations can use to communicate.

Code regions support two different approaches to search. In the first method, which we call sequential filtering, the user makes a gross stab at selecting a code region by generating all of the elements which satisfy some fairly general condition. He then sequentially restricts that set by applying more and more conditions. For example, to find "the loop which computes the sum of the test scores", he locates the set of all loops, and then restricts it to the ones which involve test scores and summations.

In the second approach, the user identifies a collection of items, possibly from several different databases, and intersects them together to find the elements which satisfy all of the conditions he wants to impose. In this "code painting" approach, the PRL combines these items essentially by overlaying the corresponding regions of code. For example, locating "the loops which compute sums" is done (figuratively) by coloring all loops red and all places that compute sums yellow. Any region which comes up orange has all of the properties that were desired.

Code painting is a deliberately coarse affair. It is designed to exploit the kind of incomplete or even slightly inaccurate information which the EPM will contain, given that much of the data is provided by the user. In some cases, code painting may not identify the exact section of the program which the user desired, but in the context of an interactive system with a screen oriented display, "close" will be good enough. To help the user see the effects of code painting, it is possible to highlight the identified section(s).

5.2 A SCENARIO USING THE PRL

The following example shows how the PRL uses the code painting paradigm to answer the question "find the initializations of the loop which computes the sum of the test scores", given the Ada program shown in Figure 4.

```
for MAXSIZE in 1..10 loop
  TOTAL:= ARRAYSUM (TEST-SCORES, MAXSIZE);
  put (TOTAL);
end loop;


function ARRAYSUM (A: in ARRAY; N: in INTEGER) return INTEGER is
  begin
    SUM: REAL:= 0;
    for I in 1..N loop
      SUM:= SUM + A(I);
    end loop;
    return SUM;
end ARRAYSUM;
```

Figure 4. The Ada Program Used in the Scenario.

In this example, the user starts by identifying three sets of data, corresponding to the summation TPPs, syntactic loops, and segmented parse frames involving the test score array.

```
> (index 'summation tpp-database)
   => TPPset1

> (index 'loops syntax-database)
   => LOOPset1:[length 2]

> (index 'TEST-SCORES segp-database)
   => SEGset1:[length 6]
```

The program only contains one TPP, but there are two loops, and several segments which relate to the variable TEST-SCORES. It is important to notice that all of these segments use the data contained in the variable TEST-SCORES but do not necessarily refer to it by that name (for example, the literal "A(I)" in the ARRAYSUM function accesses the test score array). This association is apparent from the data flow analysis within the segmented parse.

The user intersects these descriptions by invoking the code painting paradigm. The code-painting algorithm returns the largest region of text which can be described in all three ways.

```
> (overlay-code-regions TPPset1 LOOPset1 SEGset1)
   => CODE-REGION1
      **for I in 1..N loop
           SUM:= SUM + A(I);
         end loop;**
```

In order to compute this information, the overlay function automatically converts the input sets into their corresponding regions of code. Most of these translations are automatically available (though heuristic in nature). In the case of the TPP, the user had to define that mapping at some time.

At this point, the user has identified a loop which computes the sum of the test scores. In order to find the initializations of this code, he views this region from the segmented parse perspective (where initializations are represented explicitly), and scans it for segments of the appropriate type. This is a filtering operation, in which the user applies restrictions to a previously identified set of objects.

```
> (Filter (Segs-Within CODE-REGION1)
          '(Seg-Type "initialization"))
     => SEGset2:[length 2]
```

The PRL converts CODE-REGION1 to a set of segmented parse frames (a heuristic process), and the function Segs-Within enumerates the subsegments it contains. The system identifies two initializations as a result. The user prints them by converting them to the textual view.

```
> (show! SEGset2)
   => for I in **1..N** loop
   => **SUM: REAL:= 0;**
```

The answers correspond to the initializations of the iteration variable "I", and the accumulation variable, "SUM". Note that the PRL retrieves the second initialization, even though it is lexically outside of the summation loop itself. It is identified from the segmented parse analysis, which associates a loop and its initializations no matter how far apart they might have been in the original code.

6. CURRENT STATUS AND FUTURE WORK

AI&DS is now developing a prototype version of the IPE (in a three year, 2-3 person effort), which is intended to demonstrate the efficacy of our knowledge based approach to the design of programming support tools. The prototype will embody a portion of all of the facilities that have been described. The IPE is currently targeted for the Ada language. It will initially run on a Symbolics 3600, a fast, personal LISP computer that features

a high-resolution bit-map display, but it is being designed to be portable to other systems (in particular, Unix).

We expect to augment the EPM's database to include more pragmatic information (e.g., the relation between requirements and program structures), and we intend to extend the PRL to the point where it will be able to automatically plan and carry out search requests of the kind demonstrated in this paper (based on a single user query). When these extensions are complete, the PRL will define a more formal reference language.

The task of building a prototype for the IPE involves a number of issues including the incremental modification of databases, and the recognition of user intentions in code. In order to solve these problems in the context of our applied research, we expect to rely heavily on methods for eliciting information from the user, and to focus on template-oriented techniques for manipulating programs.

Acknowledgements

We would like to thank Michael Brzustowicz and Eric Domeshek for their contributions to this project.

7. REFERENCES

1. Brotsky, D., Master's Thesis, MIT, forthcoming.

2. Dean, Jeffrey S., and Brian P. McCune, "Advanced Tools for Software Maintenance", AI&DS TR 3006-1, October 1982.

3. Nelson, T., "A New Home for the Mind," Datamation, March 1982.

4. Rich, Charles, "Inspection Methods in Programming", AI-TR-604, Artificial Intelligence Laboratory, MIT, 1981.

5. Shapiro, Daniel G., Brian P. McCune, and Gerald A. Wilson, "Design of an Intelligent Program Editor", AI&DS TR 3023-1, September 1982.

6. Teitelbaum, T., T. Reps, and S. Horwitz, "The Why and Wherefore of the Cornell Program Synthesizer", Proceedings, ACM SIGPLAN/SIGOA Conference on Text Manipulation, June 1981, pp. 8-16.

7. Waters, Richard C., "Automatic Analysis of the Logical Structure of Programs", AI-TR-492, Artificial Intelligence Laboratory, MIT, 1978.

8. Waters, R., "The Programmer's Apprentice: Knowledge Based Program Editing," IEEE Transactions on Software Engineering, SE-8, 1, January 1982, pp. 1-12.

APPENDIX B

RUBRIC: A System for Rule-Based Information Retrieval

[Invited paper, COMPSAC '83, November 1983.]

RUBRIC: A SYSTEM FOR RULE-BASED INFORMATION RETRIEVAL

Brian P. McCune, Richard M. Tong, Jeffrey S. Dean, Daniel G. Shapiro

Advanced Information & Decision Systems
Mountain View, California

ABSTRACT

A research prototype software system for conceptual information retrieval has been developed. The goal of the system, called RUBRIC, is to provide more automated and relevant access to unformatted textual databases. The approach is to use production rules from artificial intelligence to define a hierarchy of retrieval subtopics, with fuzzy context expressions and specific word phrases at the bottom. RUBRIC allows the definition of detailed queries starting at a conceptual level, partial matching of a query and a document, selection of only the highest ranked documents for presentation to the user, and detailed explanation of how and why a particular document was selected. Initial experiments indicate that a RUBRIC rule set better matches human retrieval judgment than a standard Boolean keyword expression, given equal amounts of effort in defining each. The techniques presented may be useful in stand-alone retrieval systems, front-ends to existing information retrieval systems, or real-time document filtering and routing.

1. THE INFORMATION RETRIEVAL PROBLEM

The three most common approaches to textual information retrieval (see the vertices of the triangle in Figure 1), when used in isolation, suffer from problems of precision and recall, understandability, and scope of applicability. For example, Boolean keyword retrieval systems such as the commercial DIALOG system operate at a lexical level, and hence ignore much of the available information that is syntactic, semantic, pragmatic (subject-matter specific), or contextual. The underlying reasoning behind the responses of statistical retrieval systems [Salton & McGill-83] is difficult to explain to a user in an understandable and intuitive way. Systems that rely on a semantic understanding of the natural language that is present in documents [Schank & DeJong-79] must severely restrict the vocabulary and document styles allowed (e.g., to partially formatted, stereotypic messages).

In addition to being used by specialists, in the near future large on-line document repositories will be made available via computer networks to relatively naive computer users. For both classes of users, it is important that future retrieval systems possess the following attributes:

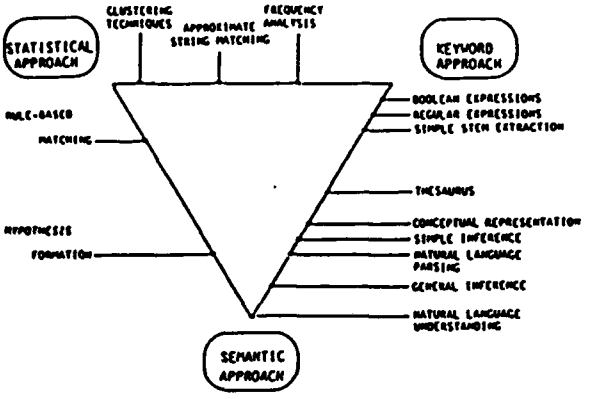


Figure 1: The Information Retrieval Triangle

- Detailed queries should be posed at the user's own conceptual level, using his or her vocabulary of concepts and without requiring complex programming.
- Partial matching of queries and documents should be provided, in order to mirror the imprecision of human interests.
- The number of documents retrieved should be dependent upon the needs of the user (e.g., uses for the documents, time constraints on reading them).
- A logical, understandable, and intuitive explanation of why each document was retrieved should be available.
- The user should be able to easily experiment with and revise the conceptual queries, in order to handle changing interests or disagreement with previous system performance.
- Conceptual queries should be easily stored for periodic use by their author and for sharing with other users.

2. A KNOWLEDGE-BASED APPROACH

In order to address the issues raised above, we have created a prototype knowledge-based full-text information retrieval system called RUBRIC (for RUle-Based Retrieval of Information by Computer). RUBRIC integrates some of the best characteristics of all three basic approaches to information retrieval (Figure 1) within the framework of a standard artificial intelligence technique. Queries are represented as a set of logical production rules that enable the user to define retrieval criteria using much better semantic and heuristic controls than can be found in current retrieval systems.

The rules define a hierarchy of retrieval topics (or concepts) and subtopics. By naming a single topic, the user automatically invokes a goal-oriented search of the tree defined by all of the subtopics that are used to define that topic. The lowest-level subtopics are defined in terms of pattern expressions in a text reference language, which allows keywords, positional contexts, and simple syntactic and semantic notions. Each rule may have a user-provided heuristic weight. This weight defines how strongly the user believes that the rule's pattern indicates the presence of the rule's subtopic. Technical issues that arise when information retrieval is viewed as a problem in evidentiary reasoning are discussed in [Tong et al.-83b].

To perform a retrieval RUBRIC uses the set of rules for a topic to create a heuristic AND/OR goal tree that defines at its leaves what patterns of words should be present in documents, and in what combinations.

Document recall by RUBRIC is enhanced by the use of higher-level notions than simple Boolean combinations of keywords. Retrieval precision is improved by the use of variable weights on each rule to define the certainty of match. These weights make it possible to present to the user only partial matches above some threshold. By tracing through rule invocation chains, an explanation facility allows the user to see exactly why a document was retrieved and why it was assigned its overall certainty or importance weight. This promotes experimentation and appropriate modification of the rule base. The retrieval vocabulary to be used is unrestricted, being left up to whoever creates the rules. Rule sets may be stored in public or private rule "libraries", so that useful subtopics may be shared among users, thus simplifying the task of defining new topics.

A rule-based query can be more complex than the keyword expression that might be used with a Boolean retrieval system. Therefore, we expect rule-based retrieval to be used initially for applications in which the same query is made repetitively over some period of time. In such situations people who are trained RUBRIC users but not programmers should be willing to expend more effort to develop a detailed rule-based definition of the query topic.

Although RUBRIC is a knowledge-based system, it really is not an expert system in the usual sense. In an expert system the system's knowledge base is an attempt to define what is known about some field of inquiry (e.g., infectious diseases, geology) in a useful form analogous to that used by human experts. Although the knowledge is never complete and perhaps not agreed upon by all experts, there exists some underlying theory or physical model that all concerned believe. In the case of information retrieval, as in other areas of preference such as politics or matters of style, there is no "right" answer. Hence, RUBRIC is really a system for capturing and evaluating human preferences. Preference systems are likely to play a much larger role in the future, as artificial intelligence tackles the problem of supporting complex, multi-attribute decision making.

3. EXPRESSING QUERY TOPICS AS PRODUCTION RULES

RUBRIC gains its power from the knowledge base of retrieval rules at its disposal. An example set of rules that defines the topic of the 1982 World Series of Baseball is given in Figure 2. These fifteen rules define a main topic, called World_Series, and a number of subtopics. The subtopics are used to define the main topic, but may also be used as query topics on their own or as subtopics of other main topics. This rule set is by no means complete; however, extensions in the form of additional rules are easy to make.

Each rule defines a logical implication; that is, the existence of the pattern on the lefthand side of the arrow ("=>") implies the existence of the topic named on the righthand side. Thus, a rule defines the topic or concept named in its righthand side. There may be multiple rules about the same topic, and RUBRIC will use each as an equally valid alternate definition (i.e., there is an implicit OR). The lefthand side of a rule is its body, which defines a pattern to be matched. This can be the topic named in the righthand side of another rule, a text reference expression (defined below), or a compound expression that defines the logical AND (denoted by "&") or OR ("|") of two or more other rule topics or text reference expressions. Explicit text to be matched without further interpretation is surrounded by quotation marks; names of topics and text reference language constructs are not. The last element in a rule is its weight, which is a real number in the interval [0,1]. It represents the rule definer's confidence that the existence in a document of the pattern defined by the rule's lefthand side implies that the document is about the topic named in the rule's righthand side. If a weight is omitted, it is assumed to be 1.0 (i.e., absolute confidence). Note that a weight is a number made up by a human user, based upon his or her experience and insight; a weight is not a statistical quantity.

```
team | event => World_Series

St._Louis_Cardinals | Milwaukee_Brewers => team

"Cardinals" => St._Louis_Cardinals (0.7)
Cardinals_full_name => St._Louis_Cardinals (0.9)

saint & "Louis" & "Cardinals"
   => Cardinals_full_name

"St." => saint (0.9)
"Saint" => saint

"Brewers" => Milwaukee_Brewers (0.5)
"Milwaukee Brewers" => Milwaukee_Brewers (0.9)

"World Series" => event
baseball_championship => event (0.9)

baseball & championship => baseball_championship

"ball" => baseball (0.5)
"baseball" => baseball

"championship" => championship (0.7)
```

Figure 2: Rule Base for Topic of World_Series

A text reference expression may be a single keyword or phrase, or a lexical context within which two keywords or phrases must be found (e.g., word adjacency, same sentence, same paragraph). So, for example, one can specify that two patterns are of interest only if they occur in the same sentence. Fuzzy (partial) matching versions of these contexts are also allowed. RUBRIC's fuzzy pattern matcher returns a value in [0,1] that is proportional to the degree that the phrases are in the desired context, i.e., inversely proportional to the logical distance between the two objects in the document. For example, when matching a fuzzy same-sentence context, two phrases in the same sentence might receive a weight of 1.0, within adjacent sentences 0.8, etc.

Rules often define alternate terms, phrases, and spellings for the same concept. Thus, rules can also provide a simple hierarchical thesaurus, with variable weights defining the degree of certainty with which a particular variant is to match. For example, in English "St." is used as the abbreviation for both "Saint" and "Street", and thus "St." is weighted less that the keyword "Saint" in Figure 2. Rules can also aid multilingual information retrieval. For example, if the database contains text in multiple languages, then the lowest level(s) of rules might define synonyms in each language of interest. The more conceptual, language-independent rules higher in the hierarchy would remain unchanged.

It has been found useful to provide a new type of rule in RUBRIC, called a modifier rule, which enables the user to incorporate auxiliary (or contextual) evidence into the query. Auxiliary evidence is evidence that by itself neither confirms nor disconfirms a hypothesis, but which may increase (or decrease) our belief if seen in conjunction with some primary evidence. The form of such a rule is

if A, then C to degree w_1;

but if also B, then C to degree w_2

where if w_1 is greater than w_2 then B is disconfirming auxiliary evidence, and if w_1 is less then w_2 then B is confirming auxiliary evidence. This has the effect of interpolating between w_1 and w_2, depending upon the certainty computed for the auxiliary clause B. Thus we might have a rule of the kind:

```
if (the story contains the literal string "bomb"),
   then (it is about an explosive device)
      to degree 0.6;
   but if also (it mentions a boxing match),
   then (reduce the strength of the conclusion)
      to degree 0.3
```

Here we see the concept of disconfirming evidence in operation; notice that by itself being about the concept boxing match is not evidence that can be used to support or deny the conclusion we are trying to establish.

Knowledge bases of rules are expected to evolve over time. Initially the set of rules provided in a knowledge base will capture a small portion of the kinds of knowledge required. New rules are easily added to RUBRIC, currently by means of a standard display-oriented text editor. Existing rules may be modified for experimentation to provide feedback for honing their logical structure, keywords, and weights.

4. QUERY PROCESSING

A set of rules defines a logical hierarchy of retrieval topics and subtopics (Figure 3). A specific retrieval request is carried out by a goal-oriented inference process similar to that used in the MYCIN medical diagnosis system [Shortliffe-76]. This process creates and evaluates an AND/OR tree of logical retrieval patterns. The root node of this tree represents a semantic topic or concept that the user wants retrieved; nodes farther down in the tree represent intermediate topics with which the root topic is defined; and nodes at the leaves of the tree represent patterns of words that are to be searched for in the database. Each arc in the tree is weighted such that the intermediate topics and keyword expressions contribute, according to their weight, to the overall confidence that the root topic has also been found. (Unlabeled arcs in Figure 3 have an implicit weight of 1.0.) Arcs representing the conjuncts of an AND expression are linked together near their common base in Figure 3.

RUBRIC supports a number of calculi for interpreting the rule weights. Weights are treated as certainty or partial truth values, not as

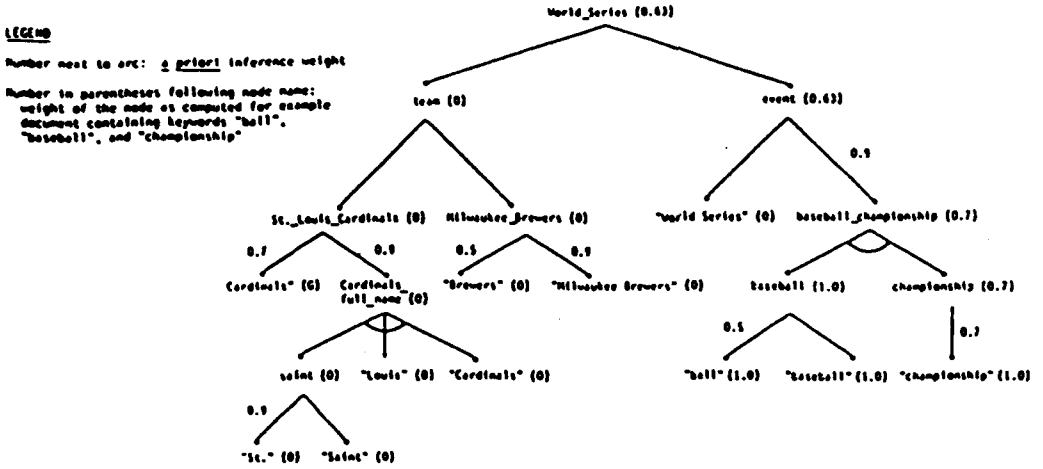


Figure 3: Rule Evaluation Tree for World_Series Topic

probabilities. Each calculus defines how to combine the uncertainties during such logical deductions as AND, OR, and implication. The default method is to use the functions minimum, maximum, and product to propagate the weights across AND and OR arcs and implication nodes, respectively [Shortliffe-76].

Referring to Figures 2 and 3, we now describe how RUBRIC processes a query. (Annotated traces of the system's operation are found in [McCune et al.-83].) When the user types in the conceptual query World_Series, RUBRIC searches its rule base for all rules that provide definitions for this topic (i.e., that have World_Series on their right-hand sides). There is only one such rule in Figure 2, so RUBRIC expands that rule according to its lefthand side. The result is that the World_Series, team, and event nodes of Figure 3 are created, as well as the two arcs between them. Since team and event are themselves the names of topics, rather than textual patterns, RUBRIC searches its rule base for their definitions. This process continues recursively until all leaf nodes of the tree contain textual patterns.

At this point each document in the database is matched against all of the phrases in the leaves of the tree. For a given document, if a phrase is found somewhere in the document, the corresponding node in the tree is assigned a value of 1.0, otherwise 0. Then the weights at the leaves are combined and propagated up through the tree to determine the overall weight to be assigned to this document.

For example, if a document contained the words "ball", "baseball", and "championship", and no other words referred to in the example rule base, then the nodes of the tree would be assigned the weights shown in parentheses in Figure 3. The "ball", "baseball", and "championship" leaf nodes all receive a weight of 1.0, and all other leaves receive a weight of 0. The baseball node would then be assigned the value 1.0 because that is the maximum of (1.0 multipled by 0.5) and (1.0 times 1.0). Similarly, the championship node receives the value 0.7. Then, because it is an AND node, the baseball_championship node gets the value 0.7, which is 1.0 times the minimum of 1.0 and 0.7. The event node then gets the value 0.63, which is the maximum of (0 times 1.0) and (0.7 times 0.9). Since there are no keywords in the document that support the team subtopic, the overall weight of the match of the World_Series topic on this document is 0.63 (1.0 times the maximum of 0 and 0.63).

Other combinations of keywords and phrases in a document can satisfy the concept of World_Series to varying degrees. Figure 4 shows the other weights possible for the World_Series topic, depending upon the dominant phrases that occur in the document.

Phrases Present in Document	Support for World_Series Topic
"World Series"	1.00
"Saint", "Louis", "Cardinals"	0.90
"Milwaukee Brewers"	0.90
"St.", "Louis", "Cardinals"	0.81
"Cardinals"	0.70
"baseball", "championship"	0.63
"Brewers"	0.50
"ball", "championship"	0.45
none of the above	0.00

Figure 4: Possible Weights for World_Series Topic

5. USER INTERFACE

A user need only see the highest weighted documents. After the database has been searched, each document that was considered has an associated weight that represents the system's confidence that the document is relevant to the topic requested by the user. RUBRIC sorts these documents into descending order based upon their weights, and groups the documents by applying statistical clustering techniques to the weights. The user is then presented with those documents that lie in a cluster containing at least one document with a weight above a threshold provided by the user (e.g., 0.8 or above). Clustering prevents an arbitrary threshold from splitting closely ranked documents. The threshold may be varied depending upon how much time the user has available to read documents, how important it is not to miss any potentially relevant ones, etc.

RUBRIC is able to explain why a particular document was retrieved. This capability is very important for instilling confidence in users and helping them get a good enough feel for the operation of the system that they can successfully write and use their own retrieval rules. RUBRIC can display each rule that results in a non-zero weight being propagated, as well as the value of that weight. RUBRIC can also show each attempt to match a word or phrase to the document, along with whether or not it matched.

6. EXPERIMENTAL RESULTS

We have done preliminary experiments with RUBRIC to examine the improvements that can be achieved over a conventional Boolean keyword approach. As an experimental database for testing the retrieval properties of RUBRIC, we have used a selection of thirty stories taken from the Reuters News Service. Our basic experimental procedure is to rate the stories in the database by inspection (i.e., define a subjective ground truth), construct a rule-based representation of a typical query, apply the query to the database, and then compare the rating produced by RUBRIC with the a priori rating.

We concentrate on two basic measures of performance. Both of these are based on the idea of using a selection threshold to partition the ordered stories so that those above it are "relevant" (either fully or marginally) and those below it are "not relevant". In the first we lower the threshold until we include all those deemed a priori relevant, and then count the number of unwanted stories that are also selected (denoted N_F). In the second we raise the threshold until we exclude all irrelevant stories, and then count the number of relevant ones that are not selected (denoted N_M). The first definition therefore gives us an insight into the system's ability to reject unwanted stories (precision), whereas second gives us insight into the system's ability to select relevant stories (recall).

We selected as a retrieval concept "violent acts of terrorism", and then constructed an appropriate rule-based query. This is summarized in Figure 5, where we make extensive use of modifier rules. An auxiliary clause is shown linked to its conclusion by a directed arc labeled "Modifier". Application of this query to the story database results in the story profile shown in Figure 6. (Notice that for presentation purposes the stories are ordered such that those determined to be a priori relevant are to the left in Figure 6). The performance scores for this experiment are

Precision: $N_F = 1$ when we ensure that $N_M = 0$, and

Recall: $N_M = 5$ when we ensure that $N_F = 0$

This is almost perfect performance, being marred only by the selection of story 25, which, although it contains many of the elements of a terrorist article, is actually a description of an unsuccessful bomb disposal attempt.

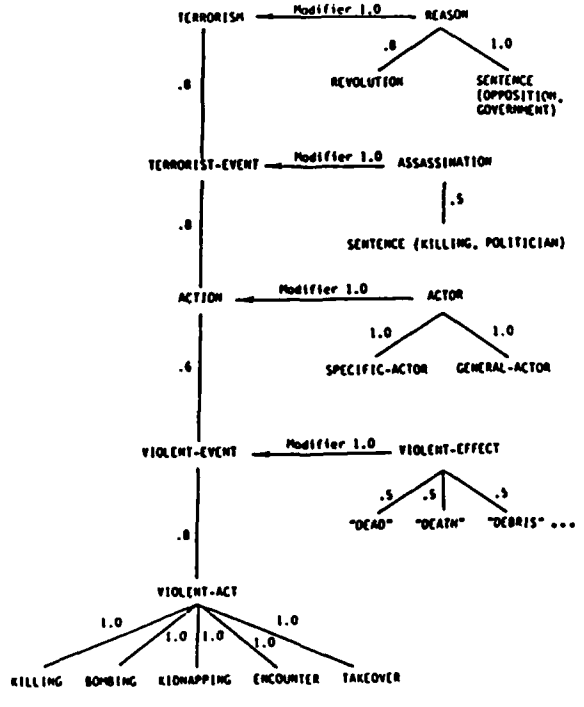


Figure 5: Rule Base Structure for Concept of Violent Acts of Terrorism

To compare RUBRIC against a more conventional pproach, we constructed two Boolean queries by ing the rule-based paradigm and setting all rule ights to 1.0 (thus incidentally showing that our thod subsumes Boolean retrieval as a special se). One of these queries is shown in Figure 7. an AND/OR tree of sub-concepts. The only fference between the two Boolean queries is that the first we insist on the conjunction of ACTOR d TERRORIST-EVENT (as shown), whereas in the cond we require the disjunction of these con- pts. The conjunctive form of the Boolean query sses five relevant stories and selects one unim- rtant story; whereas the disjunctive form selects l the relevant stories, but at the cost of also lecting seven of the irrelevant ones.

While these results represent only a prelim- ary test, we believe that they indicate that the BRIC approach allows the user to be more flexible in the specification of his or her query, thereby increasing both precision and recall. A traditional Boolean query tends either to over- or under-constrain the search procedure, giving poor recall or poor precision. We feel that, given equal amounts of effort, RUBRIC allows better models of human retrieval judgment than can be achieved with traditional Boolean mechanisms.

We have also explored the effects of using different calculi for propagating the uncertainty values within the system [Tong et al.-83A]. Among these calculi are well-known classes such as those that use "max" and "min" as disjunct and conjunct operators, and those (so-called "Bayesian-like") that use "sum" and "product". Our initial conclusion is that the calculus used is not the major determinant of performance, but that it does interact with how rules are defined.

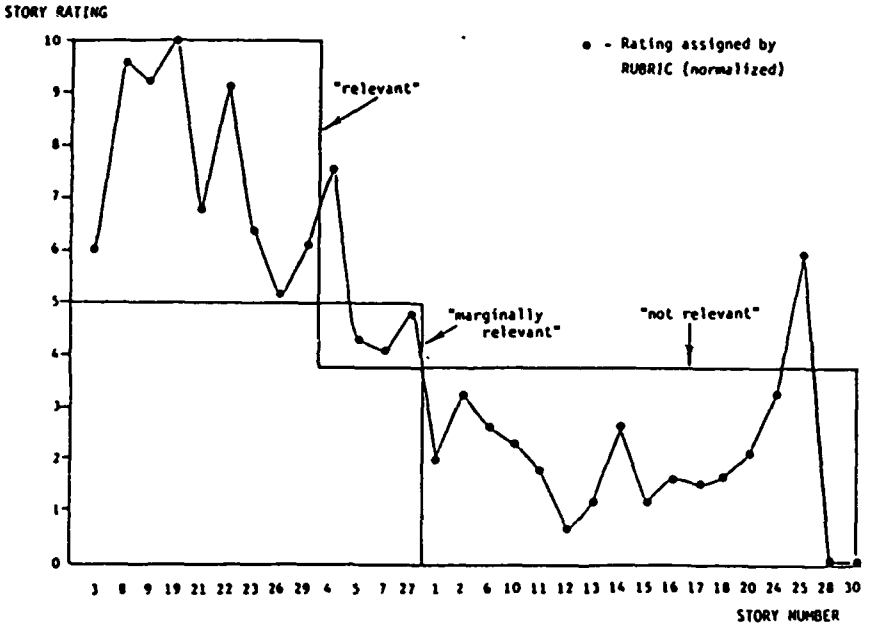


Figure 6: Story Profile from RUBRIC Experiment

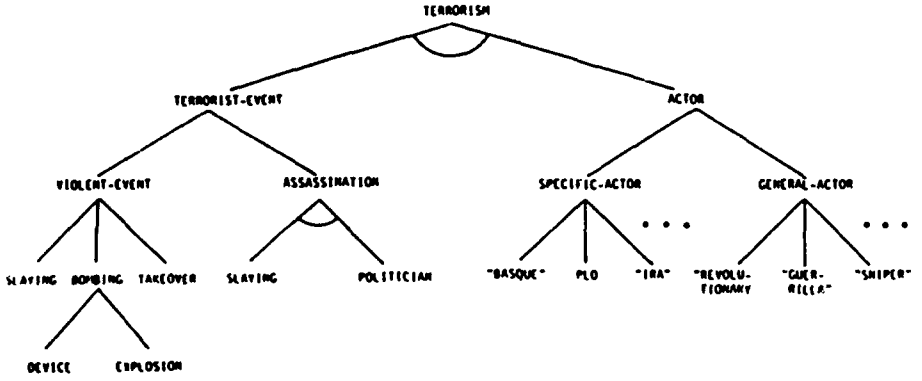


Figure 7: AND/OR Concept Tree for Boolean Query

7. FUTURE WORK

Much additional research and system develop-
t are needed to make RUBRIC usable. We are
rently providing a better user interface and
ducting more complete experiments. The inter-
e for end users will include more focused
eractive explanation, analysis of results for
sitivity to specific rules and weights, display
graphs such as Figure 6, and rule editing.
erimentation will consist of defining, in con-
ction with users, larger rule sets for a realis-
retrieval domain and then using these rules to
rieve documents from a realistic database.

Other areas of possible future work include
ing rule evaluation and textual pattern matching
e efficient, possibly through the use of heuris-
s to limit rule evaluation; exploring additional
s of representing and propagating uncertainty in
h numeric and symbolic representations; ablative
ting to measure how useful each system feature
extending the text reference language to allow
cification of the syntactic role that a word
ys in a sentence (e.g., "ship" used as a noun
sus as a verb); constructing a more general
saurus that has a network structure rather than
hierarchical one like rules; and allowing
rieval from multiple remote databases.

8. POTENTIAL APPLICATIONS

Application systems based on RUBRIC may be
ful for information routing and change detec-
n, in addition to information retrieval. For
ormation retrieval RUBRIC could be extended to
t on formatted documents such as messages or
liographic entries, to work as a front end to
xting databases and information retrieval sys-
, and to segment larger documents by subtopics.
IC could be used to process messages in real-
, filtering the important ones and routing them
he appropriate recipient (human or another pro-
). With RUBRIC, analyses of documents over
could detect statistical changes at a concep-
level rather than just in the use of indivi-
keywords.

9. REFERENCES

[McCune et al.-83] Brian P. McCune, Jeffrey S. Dean, Richard M. Tong, and Daniel G. Shapiro, RUBRIC: A System for Rule-Based Information Retrieval, Technical Report 1018-1, Advanced Information & Decision Systems, Mountain View, California, February 1983.

[Salton & McGill-83] Gerard Salton and Michael J. McGill, Introduction to Modern Information Retrieval, McGraw-Hill Book Company, New York, New York, 1983.

- [Schank & DeJong-79] R. C. Schank and G. DeJong, "Purposive Understanding", Chapter 24, in J. E. Hayes, D. Michie, and L. I. Mikulich, editors, Machine Intelligence, Volume 9, 1979, pages 459-478.

- [Shortliffe-76] Edward Hance Shortliffe, Computer-Based Medical Consultations: MYCIN, American Elsevier Publishing Company, Inc., New York, New York, 1976.

- [Tong et al.-83A] Richard M. Tong, Daniel G. Shapiro, Jeffrey S. Dean, and Brian P. McCune, "A Comparison of Uncertainty Calculi in an Expert System for Information Retrieval", in Alan Bundy, editor, Proceedings of the Eighth International Joint Conference on Artificial Intelligence, William Kaufmann, Inc., Los Altos, California, August 1983, Volume 1, pages 194-197.

- [Tong et al.-83B] Richard M. Tong, Daniel G. Shapiro, Brian P. McCune, and Jeffrey S. Dean, "A Rule-Based Approach to Information Retrieval: Some Results and Comments", Proceedings of the National Conference on Artificial Intelligence, William Kaufmann, Inc., Los Altos, California, August 1983, pages 411-415.

END

FILMED

7-85

DTIC